Hardware and Software based optimizations for Memory Address Translation

CS 694: Seminar

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology
Computer Science and Engineering

by

A Asish (23M0759)

under the guidance of

Prof. Purushottam Kulkarni
Prof. Biswabandan Panda

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400 076

Spring 2024

Acknowledgement

I would like to thank Prof. Purushottam Kulkarni and Prof. Biswabandan Panda for their suggestions, guidance and for giving me this opportunity to work under them, which made it possible for me to prepare this seminar report. I had a great experience full of learnings during this process.

Contents

Abstract

Hardware and software being the two entities of a computing system, are often viewed as disjoint. Instead, the opposite is true actually, and they both have been used together in many areas to provide powerful mechanisms and optimizations.

One of such areas is memory address translation, which would be the focus of this seminar. Memory address translation is made possible through co-operation between operating system which is software, and memory management unit which is hardware. First we would be looking at prerequisites to understand this report and current memory address translation mechanisms. Then we would explore ways to improve them, such as by reducing TLB misses, reducing number of page table levels, re-designing the indexing mechanism in page tables, and migration and replication of page tables in NUMA systems. Finally scope of future work in this area would be discussed.

Chapter 1

Introduction

Hardware and Software are two essential units of any computing system. They are sometimes viewed as decoupled and unrelated to each other but are in-fact highly related to each other and are often used in conjunction in many cases to provide some of the most important functionalities and optimizations.

We are going to talk about one of such areas in this report, called memory address translation. Memory address translation is the process of translating a virtual address to a physical address, and is possible through co-ordination between page table and operating system, which are software entities, and memory management unit which is hardware entity. Current address translation mechanisms, even after being highly researched and optimized, are unable to catch up with the rapid improvements in processor speeds, as a consequence of which memory address translation has become one of the major bottlenecks, taking upto 50% of application execution time in worst cases [1].

First we would be looking at prerequisites to understand this report. We would also be looking at current memory address translation mechanisms using radix page tables on physical machines, and nested radix page tables on virtual machines. Then we would explore ways to improve them, such as by reducing TLB misses, reducing number of page table levels, re-designing the indexing mechanism in page tables. We would then touch upon NUMA systems and discuss methods to improve efficiency of address translations on them using page table replication and migration. We would be concluding by deliberating about scope of future work in this area.

Chapter 2

Background

2.1 Operating system concepts

Memory address translation is a consequence of the mechanisms current operating systems use to provide process isolation and abstraction. To understand memory address translation, we need to first understand some key OS terms and concepts mentioned below:

1. Process : To execute a program on a processor, the code of the program is first loaded into the main memory, and then the appropriate register values are set in the processor to start executing the program. Such a program which is in main memory along with it's runtime data like stack, heap, register values is called a process. Multiple processes can reside in main memory and execute concurrently using the mechanism of context switch.

2. Address space abstraction : Main memory is divided into chunks of certain size called pages. Usually pages are of size 4KB, 2MB and 1GB on x86 systems. Modern OSes use the mechanism of virtual addressing to abstract out the physical memory from a process. Each process has it's own virtual address space, which is just a sequence of addresses starting from address 0 upto the size of that address space. Process can only generate and access these virtual addresses, but for operations to actually happen, these virtual addresses need to be mapped to some physical addresses.

3. Page Table : Page table (PT) is a software data structure that has translations from virtual addresses to their corresponding physical addresses. Each process has it's own page table which map addresses from it's virtual address space to

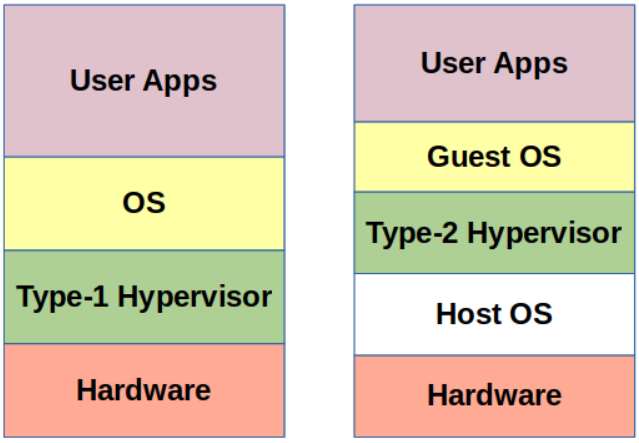

Figure 2.1: Hypervisor Types

physical memory. When a process is about to run, it's page table is loaded/set-up appropriately by the OS for the process to use.

Just like address space has been abstracted from a process, it is possible to abstract entire hardware from a machine, through a mechanism called virtualization.

2.2 What is virtualization?

Virtualization is a mechanism to abstract the physical hardware from OS and provide various virtual entities like virtual CPUs, memory, IO devices. A complete virutalized setup with a host OS is also called a virtual machine (VM). The software that manages virtualization is known as a Virtual Machine Manager (VMM) or a hypervisor. Hypervisor can be working on the underlying hardware itself (Type 1) or maybe installed as a program on the host OS (Type 2), as shown in figure 2.1. In any of the case, the functionality of hypervisor is to manage virtualization of hardware resources.

2.3 Overview of page tables

2.3.1 Tree based page tables

The following are page table designs structured like a tree :

1. Radix page table : Radix page table is a way of organizing the page table like a radix tree (trie) in a multi-level fashion. Virtual address (VA) is divided

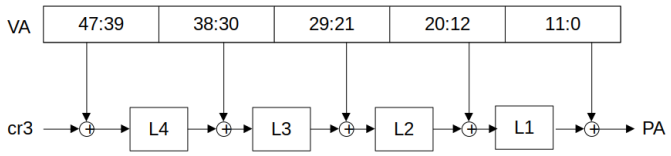

Figure 2.2: Radix page table on x86-64

into groups of bits and successive groups of bits are used to find the correct page in each level for a given VA to find the required VA to physical address (PA) translation.

Consider a 48 bit virtual address space, 32 bit physical address space and assume 4KB page size. Each physical address will be of 32 bits or 4B, which would mean one page can store upto 512 physical addresses. First 9 bits of the 48 bit VA are used to index into the uppermost level of PT to find the entry which contains the physical address of the page of next level. Then next 9 bits are used to move to the next level similarly. After moving through all the four levels of page table, we get the target physical page/frame number (PFN). Last 12 bits of VA as an offset are added to PFN to get the required physical address. This process is called page table walk. On x86 machines, page table usually has 4 levels L4, L3, L2 and L1 called Page Global Directory (PGD), Page Upper Directory (PMD), Page Middle Directory (PMD), and Page Table Entry (PTE) respectively [1]. Page table walk may require upto 4 memory accesses. The page table walk described above has been shown in figure 2.2.

2. Nested Radix page table : With virtualization, there are two page tables to walk : guest page table to translate from guest virtual address to guest physical address, host page table to translate from guest physical address(gPA) to host physical address (hPA). Figure 2.3 shows a nested page table [2] walk. Guest page table has 4 levels namely gL4, gL3, gL2 and gL1, similar to host page table. Hardware registers hcr3 and gcr3 contain the address of top levels of host and guest page tables respectively. Each guest page table page is actually in host physical memory, but guest PTEs and gcr3 contain only gPAs and not hPAs. So to obtain hPA of any gPA, host page table is walked. This process is repeated for translating the PTE entries at each guest page table level. Upto 24 memory accesses may be required for an address translation in a nested radix page table.

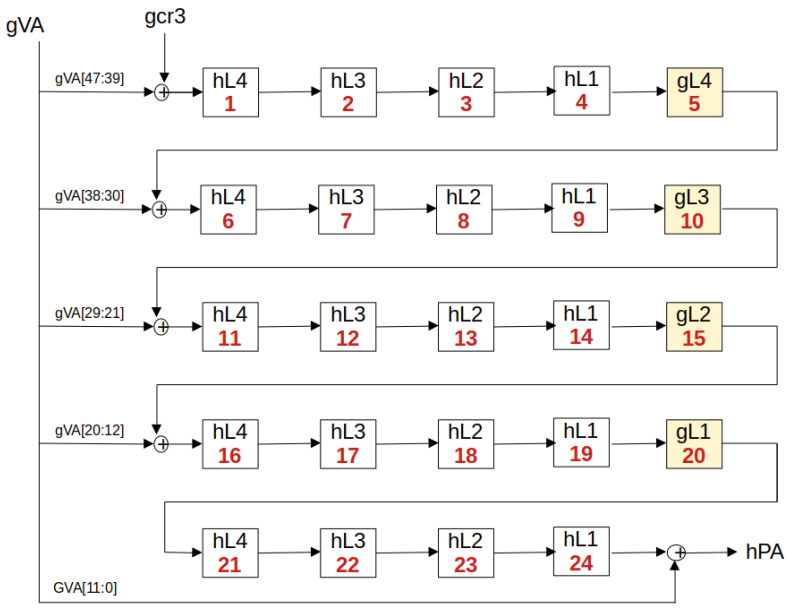

Figure 2.3: Nested radix page table on x86-64

2.3.2 Hash based page tables

Below are the page table designs that use hashing for address translation :

1. Hashed Page Table (HPT) : Hashed page table [1] is a type of page table where the virtual page number (VPN) is hashed using a hash function to generate an index of HPT. The index is then added to base address of HPT to find the exact entry in HPT that contains the physical frame number(PFN), which completes the translation. Translation using HPT may require only one memory access in case of no collisions, as shown in figure 2.4. In case of collisions, collision resolution techniques like chaining and probing may be used to find the required translation.

2. Nested Hashed page table : Hardware registers gcr3 and hcr3 contain the base addresses of guest and host hashed page tables respectively. First we obtain guest page table entry (gPTE) by hashing guest virtual address (gVA) using guest hash function (gH), which gives us the gPA of that entry. To find the gPTE using the gPA we got in host physical memory, we hash it using host hash function (hH) to find it's hPA in host page table entry (hPTE). Now we actually get the entry that holds the gPA translation of the gVA, and to convert this to hPA we again hash using hH to get the required translation. Upto 3 memory accesses may be required for address translation [2]. The

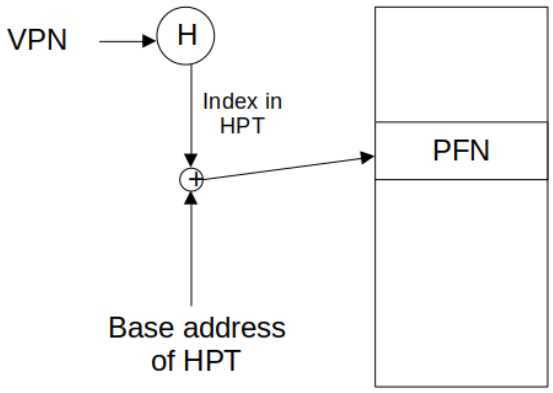

Figure 2.4: Hashed page table

nested translation process described above has been shown in figure 2.5.

Cuckoo Hashing : It is a type of hashing where we have multiple hash tables (called 'multiple ways') and may have different hash function for each hash table [2]. To insert a value, a way is chosen and hash value calculated using the hash function of that way to find the index to insert the value at. If the index is occupied, the current occupant at that place is evicted, the current value is inserted, and the evicted value is hashed for hash table of some other way. This process keeps on repeating till all values are not inserted, or maximum tries to insert are reached, when it is called an insertion failure and values in all tables are rehashed with different hash functions to overcome it.

2.4 Hardware

Hardware is a crucial part of address translation mechanism. Given below are a few hardware entities that are present in Memory Management Unit (MMU) of the processor and play important role in address translation :

1. Page Walk Caches (PWC) : Page Walk Caches are hardware caches. Each level of page table has a PWC that is used to cache the frequently occuring intermediate virtual to physical translations. PWCs reduce the time taken for a page table walk as access time of page walk cache is around 1ns, which is significantly lower than the access time of main memory that is about 50ns.

2. Translation Lookaside Buffer (TLB) : TLB is a per-core hardware cache that stores the recent virtual to physical address translations. When a virtual

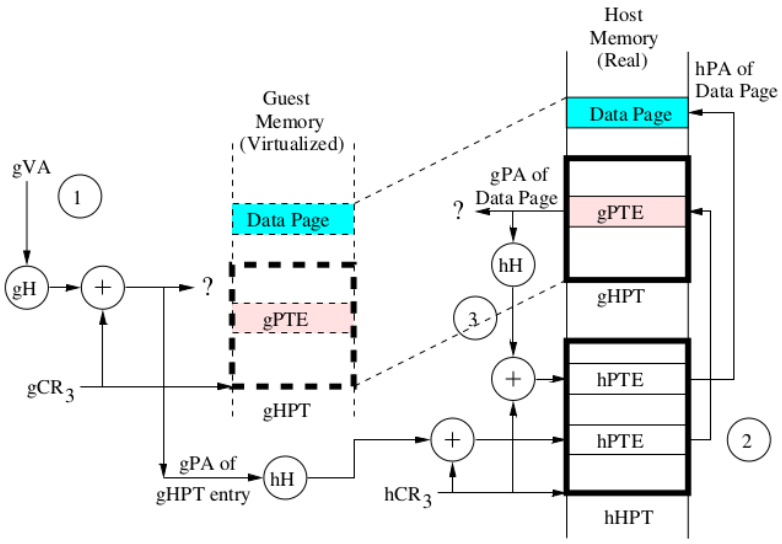

Figure 2.5: Nested hashed page table [2]

address is generated, first TLB is checked for the required translation, which takes just about 0.25ns for accessing. A TLB miss triggers page table walk.

2.5 Memory hierarchy

When paging is enabled and the processor is in protected mode (which is the general case), processor uses a virtual address whenever it requires a memory access. The address translation task is taken up by the MMU (Memory Management Unit).

First the TLB (if there may be multiple TLBs, then all are looked up) for the translation. If the translation is not found in TLB, page table walk is triggered, which gets the required translation. If the translation is not present even in the page table (which means the page we require is not in main memory and so the virtual address is not mapped to a physical address in main memory), it leads to page fault, which then triggers the OS to bring in the required page from some secondary storage into the main memory and update the page table as required.

Level 1 (L1) cache is accessed in parallel to TLB. If the translation is found not in TLB, the physical address bits are used to check tags in the relevant cache line of L1 cache. If we find the required page in L1 cache, it is accessed.

L2 cache is accessed when we get a miss in L1 cache. If we don't find the required page even in L2 cache, L3 cache is checked similarly. In case of a L3 cache miss, the data is fetched from main memory.

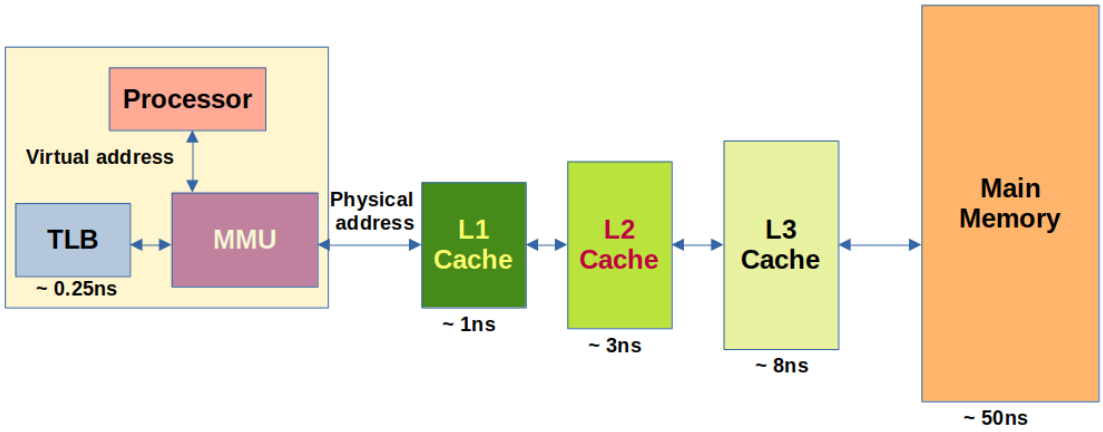

Figure 2.6: Memory Hierarchy with access times of different units

The memory hierarchy with respective latencies (in nanoseconds) of various hardware components has been shown in figure 2.6, considering a 4GHz processor.

2.6 Non Uniform Memory Access

Non-uniform memory access (NUMA) is a type of organization where the system is divided into multiple sockets with each socket having it's own cores, cache hierarchy and local main memory. The sockets are connected to each other through cache-coherent interconnect with their memories collectively forming global main memory. Access to memory is non-uniform, with the access to local memory from a core having lesser latency (upto 2x to 4x faster) as compared to memory on a remote socket [3].

NUMA is highly used in data centers to run multi-threaded applications or servers which have a large memory footprint beyond the capacity of a single socket, on multiple sockets. It is also used by cloud providers who have a large resource capacity and therefore the resources are divided among multiple sockets for better resource management. NUMA provides the flexibility to handle individual sockets independently, while also providing fault tolerance where a multi-socket application can keep on running even in case of failure of some socket.

Figure 2.7 shows a NUMA system with 2 sockets (socket 0 and socket 1). Both sockets have their own local memory hierarchy and set of processors, and are connected to each other through a cache coherent interconnect.

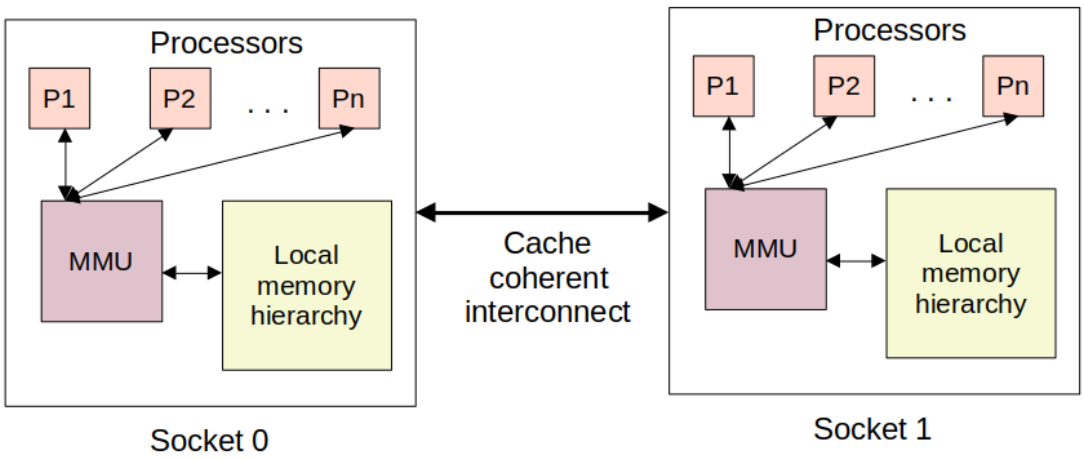

Figure 2.7: A NUMA system with two sockets connected by a cache coherent interconnect

Chapter 3

Don't Miss the TLB!

The TLB, as seen earlier, is an important part of address translation setup. When the required virtual to physical mapping is found in TLB, it takes just about 0.25 ns for address translation process, which is much faster as compared to the case when translation is not found in TLB, which then triggers a page table walk that may take upto 4 memory accesses (about 50 ns for each memory access).

A trivial way of reducing the address translation overhead is to reduce the number of TLB misses itself. One way of achieving this is by using huge pages instead of base pages. Huge pages (usually 2MB or 1GB) cover more area of memory as compared to base pages (4KB), which would mean more memory coverage per TLB entry, hence increasing chances of TLB hits. But in a typical system with many processes running concurrently, it is not possible to use only huge pages because of various reasons like :

(i) Limited availability of memory may lead to inability of huge page allocation, especially when memory is highly fragmented.

(ii) Huge pages lead to internal fragmentation, especially when the process is small. This excerbates the situation when system is already under memory pressure.

(iii) Huge page allocation may take long time (upto 90 seconds [4]) as the system needs to first find contiguous block of physical memory of huge page's size.

This calls for a balanced approach, where we allocate huge pages in a limited number as and when required. Linux already provides an abstraction called Transparent Huge Page (THP) which automates creation, management and use of huge

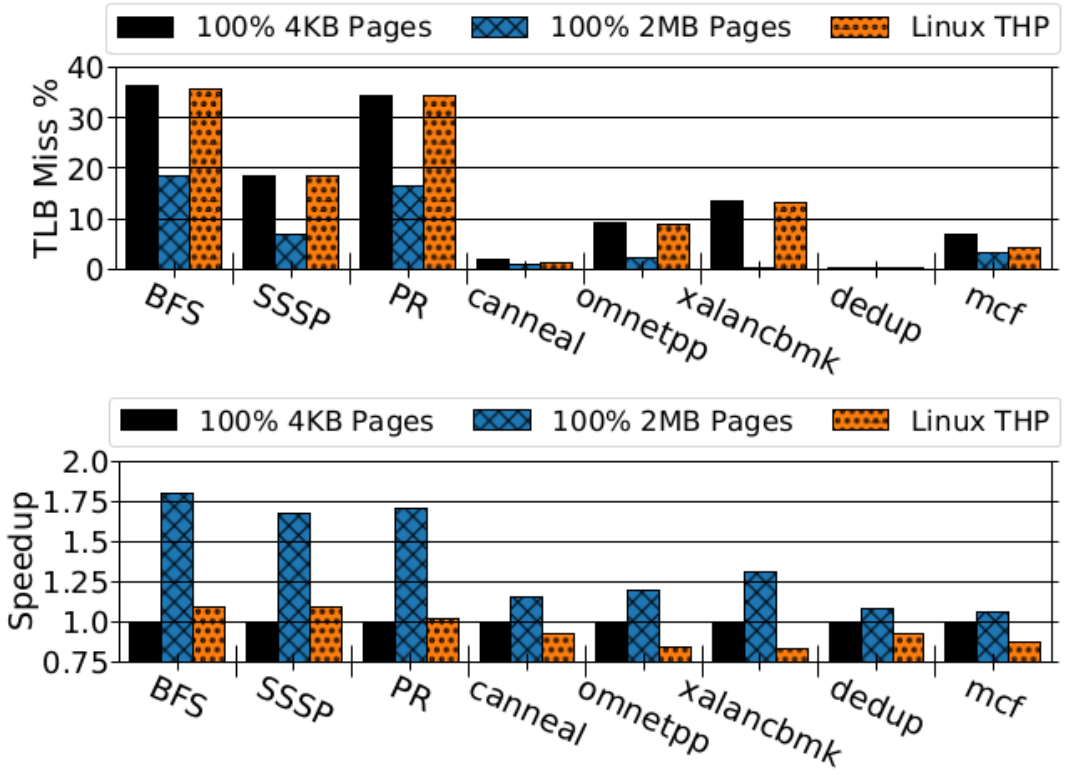

Figure 3.1: Comparison of TLB miss rates and speedups when using base pages alone, huge pages alone and THP [4]

pages. It also has settings which can be configured using madvise system call (using MADV_HUGEPAGE flag) that allow user to decide the regions to promote into huge pages [4]. But this would make things complicated for user.

Figure 3.1 shows the TLB miss percentage and speedup of several applications on using 4KB pages only, 2MB huge pages only, and Linux THP. As we can see, ideal performance is on using huge pages alone (lowest TLB miss percentage and highest speedup), whereas THP doesn't provide any substantial improvement over using base pages alone. This is because THP lacks knowledge of application behaviour and hence can't classify pages according to TLB sensitivities. Promoting regions into huge pages efficiently requires us to identify regions of memory which are best to be promoted as huge pages, given that we can allocate only limited number of huge pages. Memory regions are first classified into various categories to find out the best regions to promote and which regions need not be promoted at all, as proposed in [4].

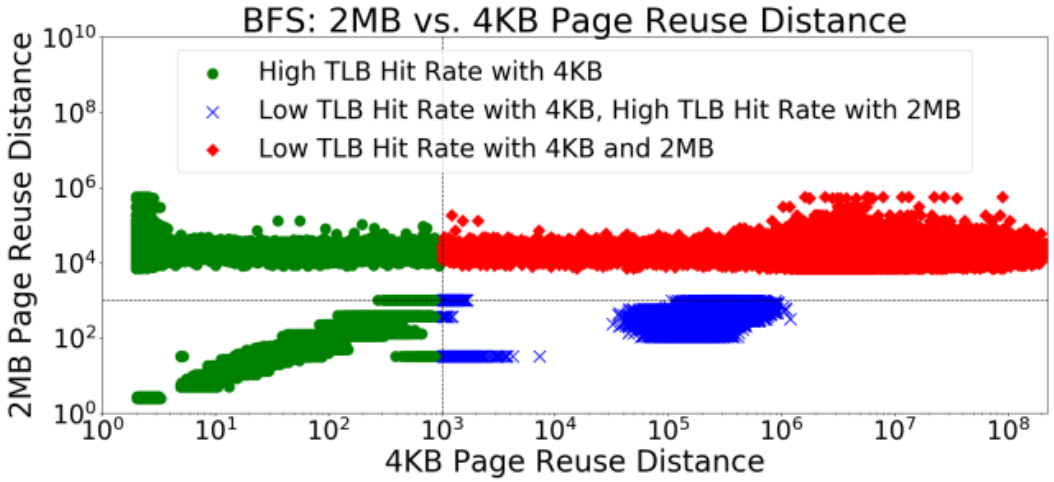

Figure 3.2: Classification of memory accesses based on RUD with 2MB vs 4KB pages, running BFS on a Kronecker network [4]

3.1 Classification of memory accesses

Page Reuse Distance(RUD) of a page is defined as number of accesses to other pages before accessing the same page again. Memory accesses are classified into three categories based on RUD :

(i) TLB friendly accesses : These are the accesses with low RUD (< 1024) at base page size (4KB). They show high spatial and temporal locality.

(ii) High-Reuse TLB sensitive accesses (HUBs) : They have high RUD at 4KB page size but low RUD at 2MB page size, i.e, they have good locality of reference at 2MB granularity.

(iii) Low-Reuse TLB sensitive accesses : They have high RUD at all page sizes, i.e, random memory access pattern.

Figure 3.2 shows the RUD at 2MB page size on y-axis and RUD at 4KB page size on x-axis for different types of memory accesses. It can be seen that TLB friendly accesses already hit in TLB at high rate at 4KB page size, so they needn't be promoted. Also, it is not useful to promote memory regions having Low-reuse TLB sensitive accesses as they mostly miss TLB even at 2MB page size. So clearly it is most beneficial to promote HUBs to huge pages as they miss TLB at 4KB page size but their TLB hit rate is high at 2MB page size [4].

3.2 Efficient huge page promotion

Promotion Candidate Cache (PCC) is a novel hardware proposed in [4], to track the per-huge page region page table walk frequency (the number of TLB misses or number of times page table was walked in each huge page region). We need to track this to know which huge page regions are having highest TLB misses when 4KB pages are being used in that region and thus would benefit from getting promoted to a huge page.

PCC is a small, fully associative hardware. PCC doesn't track TLB friendly accesses, as it is placed after the last level of page table, enabling it to track only regions that incur page table walks, which the TLB friendly access regions don't as they hit in TLB already. It also avoids cold misses(a page that is being accessed first time will miss TLB, and such a miss is called cold miss) by tracking data of only those pages whose access bits are set. It has tag as virtual address prefix (VPN), which tells the memory region being tracked, and the corresponding data entry is the frequency of page table walks in that region. There can be two different PCCs to track 2MB and 1GB huge page regions separately.

Figure 3.3 shows a flow-chart that explains how PCC works as described below:

(i) Whenever a PT walk happens and access bit of a huge page region is set, corresponding PCC is accessed.

(ii) If the region is present already in PCC, it's PT walk frequency is incremented, else a new entry is inserted with it's tag along with zero frequency.

(iii) In this process, if PCC is found to be full, some existing entry is evicted before new insertion according to PCC replacement policy which is configurable.

Figure 3.4 shows the steps of PCC working with a memory hierarchy. There are two PCCs : one to track 2MB huge page regions and another to track 1GB huge page regions. If promotion to 2MB pages is not enough and still high TLB miss rate is prevailing in a 1GB region, it can be promoted to a 1GB huge page using the data from 1GB PCC.

3.2.1 OS integration

Data from PCC can be dumped into a predetermined memory region periodically for the OS to read from, or OS itself can read data from PCC periodically to promote some N huge page regions. PCC itself can sort the regions by their frequencies,

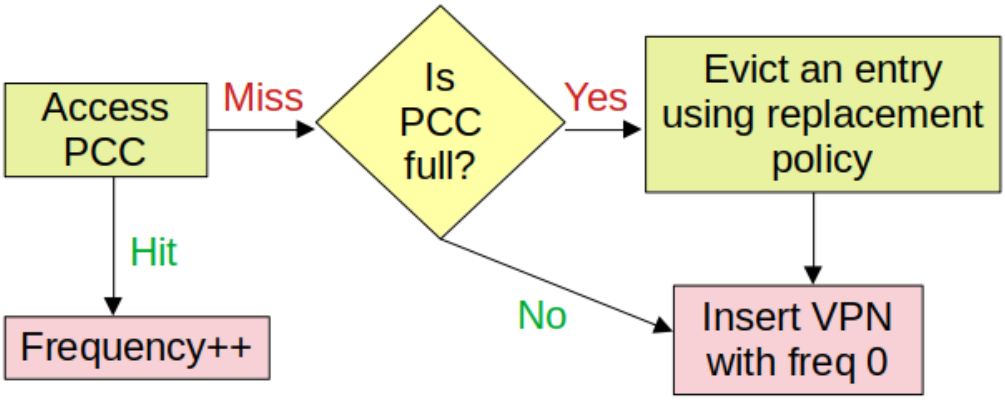

Figure 3.3: PCC operation

or OS can take up sorting if PCC has to be implemented in simple way. While promoting, OS invalidates those entries from PCC.

This decoupling of tracking of huge page region data and huge page promotion provides great flexibility in terms of promotion frequency, ability to give preferential treatment to certain regions or process, and promotion policy.

3.2.2 Performance analysis

PCC provides 1.19-1.33x speedups over 4KB base pages while backing only 1-4% of application data with huge pages.

Figure 3.5 shows the speedup and page table walk (PTW) percentage for PCC, HawkEye, THP with 50% and 90% memory fragmentations as well as maximum performance of THP on different workloads . As we can see, PCC outperforms HawkEye, which is a software approach to this huge page promotion problem. There are several reasons for this :

(i) PCC tracks per huge-page region PTW frequency, which is more relevant metric as compared to per huge-page region number of base pages accessed (number of unique 4KB pages accessed within a huge page region) which HawkEye tracks.

(ii) Classification of memory accesses and tracking only HUBs is another thing that HawkEye doesn't do and so it needs to track entire memory.

(iii) HawkEye being a software approach adds significant overhead on OS to track the data. To tackle that, HawkEye is run periodically only for a specific period

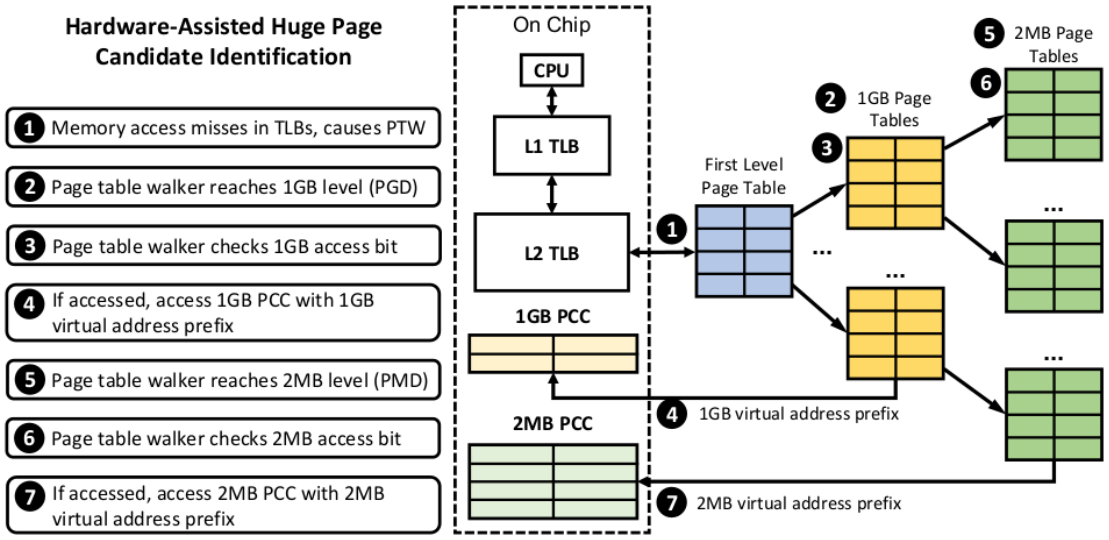

Figure 3.4: PCC overview [4]

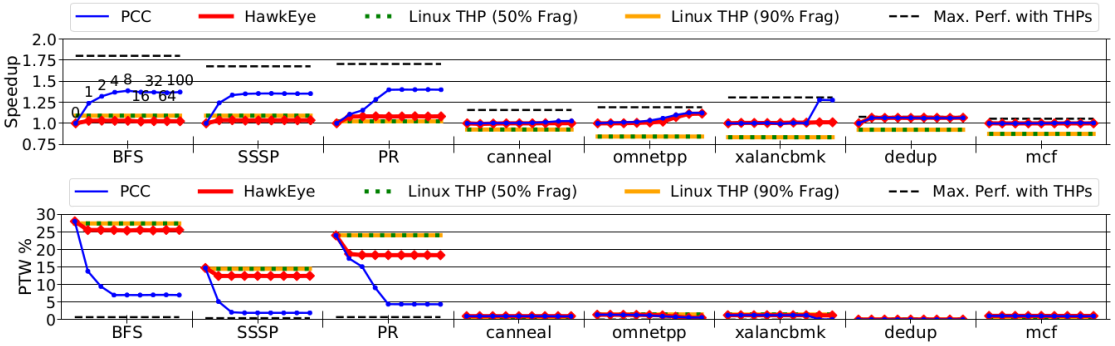

Figure 3.5: Single thread runtime performance and PTW rate comparison [4]

of time during which it is able to scan only 8 huge page regions, therefore being unable to make accurate decisions and perform many promotions. PCC on the other hand offloads the burden of tracking data about huge page regions from OS and is not restrained by any time period as it is in operation always.

PCC support to provide a mechanism for efficient huge page promotion is thus an excellent example of how hardware support can be extremely useful in implementing these types of functionalities in systems efficiently. We could see that it outperforms the existing Linux THP as well as software approach like HawkEye because of shifting the overload of huge page region tracking from software to hardware. Reducing TLB misses is also shown to improve efficiency of memory address translation as seen from application speedups.

Chapter 4

Reducing page table levels

Today main memory size is increasing at a rapid pace, and it has become impossible for TLB to catch up with that. TLB has to remain small for retaining it's speed, but the working set of applications has become much larger than what can fit inside a TLB, which makes frequent page table walks inevitable. So, address translation overhead reduction has to come from page table walk itself, rather than TLB alone.

Page table walk in a radix page table may take upto 4 main memory accesses, as shown in chapter 2. We can reduce the levels of page table that need to be walked by placing data on huge pages. But as explained in chapter 3, it is not possible to place all data on huge pages.

The situation is worse in case of virtualization, where in a nested radix page table, page table walk may require upto 24 memory accesses, as shown in chapter 2. But unlike in native system where we couldn't take advantage of huge pages to reduce levels of page table, virtualization here provides us with a unique idea proposed in [5].

4.1 Shortening nested page table walk

HugeGPT is an approach proposed in [5], to reduce the number of levels to walk in nested radix page table. The main idea is to store the entire guest page table on host huge pages, to eliminate the last level of host page table from the nested page table walk and reduce the number of potential memory accesses needed from 24 to 20, as shown in figure 4.1.

This idea is feasible due to the fact that page tables are much smaller than the process itself, so we can have enough huge pages to store page table on huge pages, even if we can't store the entire process on huge pages. This approach also gives

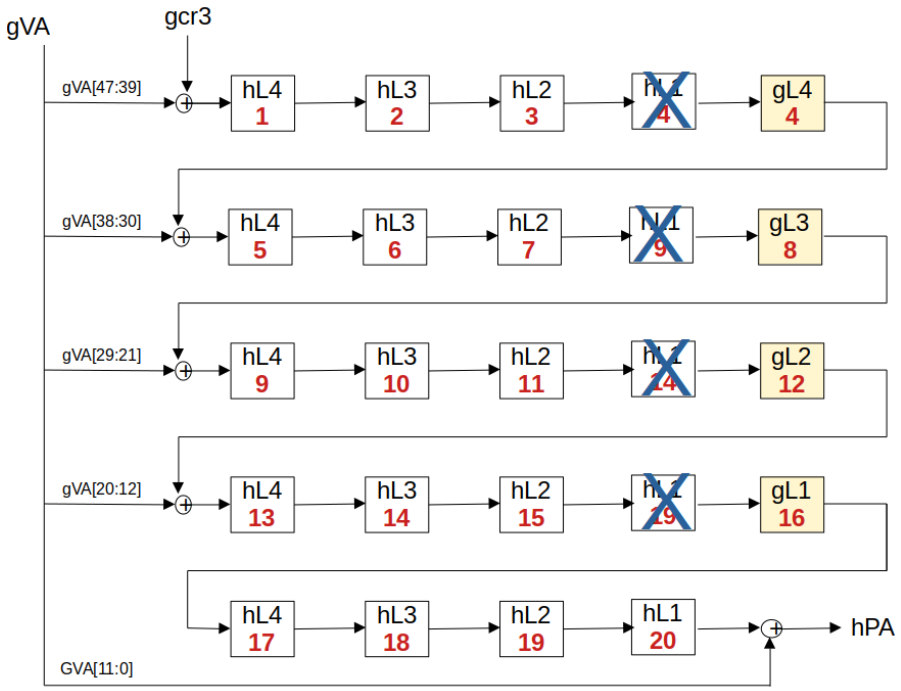

Figure 4.1: Nested page table walk with HugeGPT

an added advantage in case of workloads with weak memory access locality, where removing the last level of page table benefits such workloads and ensures better caching of translations in page walk caches. This leads to reduction in page walk cache misses and hence less page table walk latency.

4.2 Implementation

4.2.1 System initializaion

HugeGPT guest PT memory allocator first pre-allocates several huge page sized regions from the default guest memory allocator. These regions are used to store guest page table. Guest memory allocator informs the addresses of these regions to huge page requester in host. In guest OS, kernel functions used to allocate (pte_alloc_one) and free (free_pmds) page table pages have been modified to pass requests to allocator. When a request is sent, page table memory allocator returns free pages from the pool of pre-allocated regions in guest.

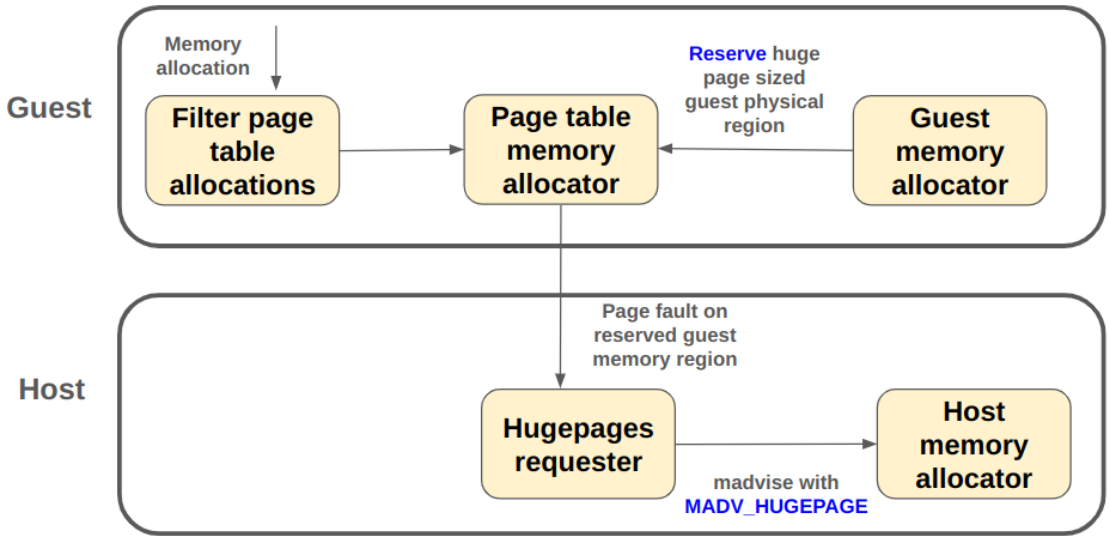

Figure 4.2: Working of HugeGPT

4.2.2 Working steps

Working steps are shown in figure 4.2 and are mentioned below :

- Allocation requests of page table are filtered by a filter and sent to memory allocator to get guest physical pages to store the page table on.
- Memory allocator issues a page fault with a pre-allocated huge page region in guest memory.
- Huge page requester in host sends madvise request with MADV_HUGEPAGE command so that host allocates a huge page to back the faulted guest memory region.
- Memory allocator won't issue more page faults till this entire region is used up to store guest page table pages.
- If all pre-allocated regions are exhausted, initialization process is run again to get more such huge page sized regions.

4.3 Performance analysis

HugeGPT provides 10% more throughput compared to Linux/KVM for throughput oriented workloads, as can be seen in figure 4.3.

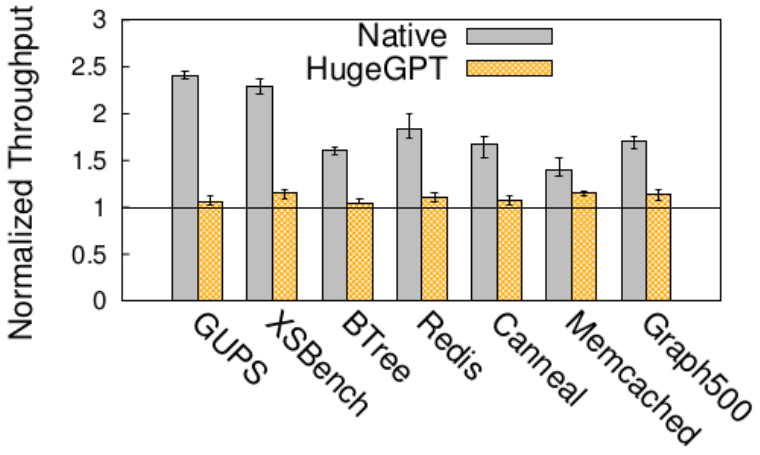

Figure 4.3: Comparison of throughput for HugeGPT, Linux/KVM and native system [5]

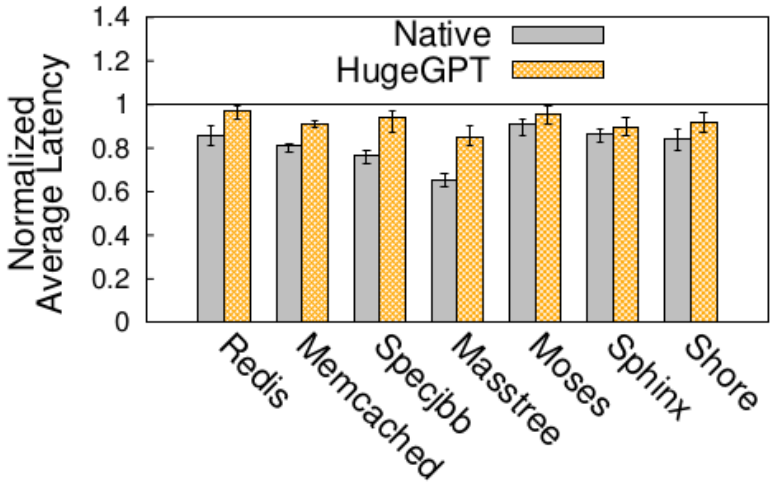

Figure 4.4: Comparison of latency for HugeGPT,Linux/KVM and native system[5]

For latency oriented workloads, HugeGPT reduces average latency by 8% compared to Linux/KVM, as can be seen in figure 4.4.

As we saw, the performance improvement is marginal. It is because the number of potential memory accesses is still 20. The problem of multi-level page table walk still remains, which motivates us to think about redesign of page table itself to something that is not level-wise radix tree like structure.

Chapter 5

Restructuring the indexing mechanism

We have seen ways to reduce TLB misses in chapter 3 and to reduce number of levels of radix page table in chapter 4. But the problem still persists, due to facts that TLB size is not scalable and page table walk in radix page table still remains sequential, unable to take advantage of memory level parallelism. With growing working set of applications, some architectures have introduced one more level of page table [1], taking the number of levels to 5. This number is bound to only grow further as memory footprints of applications keep growing. Therefore there needs to be a new design of page table which is not organized like a radix tree.

Hashed page tables are one of the natural substitute of radix page table to think about. As shown in chapter 2, they use hashing to find required page table entry in a single step when there are no collisions (which is an ideal scenario).

To show the significance of collisions, a global hash table with enough space for translations needed by all applications was taken with BLAKE Cryptographic function as hash function, which minimizes the probability of collisions [1]. Figure 5.1 shows probability of keys mapping to same entry in hash table in two different cases: when table is of enough size for all applications (only 35% keys don't collide), and when table size is 1.5x the number of entries (only 50% keys don't collide). This shows that collisions are serious issue in hashed page tables, which trigger collision resolution techniques like chaining and probing, that take additional memory accesses. Hashed page tables also don't have efficient way of resizing dynamically, which is required to avoid statically allocating entire table at beginning of process which may not be feasible. To address these drawbacks, [1] proposes a new design of page table.

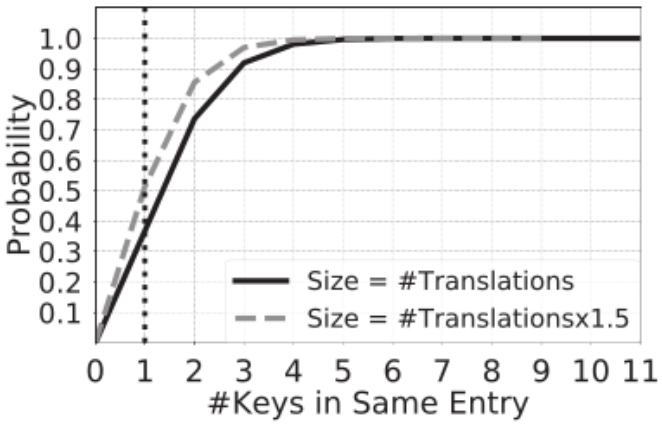

Figure 5.1: Cumulative distribution function showing number of keys that are mapping to the same entry in hash table [1]

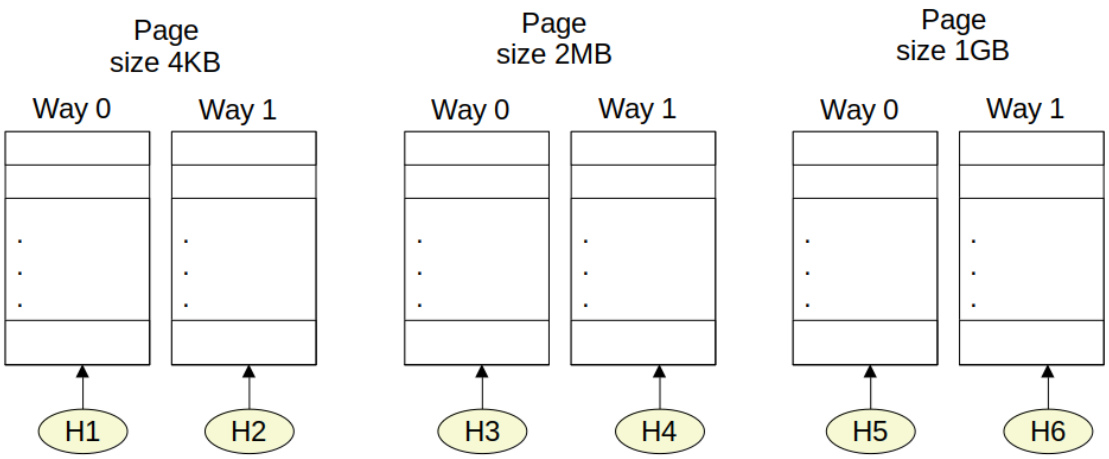

Figure 5.2: 2-way ECPT for 3 page sizes

5.1 Elastic hashed page table

Elastic Cuckoo Page Table (ECPT) [1] is an efficient and elastic hashed page table. It uses cuckoo hashing and a resizing algorithm to overcome the challenges of hashed PTs. Each process has d-ary ECPTs for each page size (PUD, PMD, PTE). Figure 5.2 shows a 2 way ECPT for 3 page sizes : 4KB, 2MB and 1GB.

Hashing based approach has another disadvantage of scattering consecutive page table entries. Also, hashed page table needs additional space to store the VPN tag with each entry. This situation can be improved by using earlier concepts compaction and clustering. ECPT uses them too, by storing VPN tag within extra bits given by linux in each PT entry for experimental purposes, as well as clustering

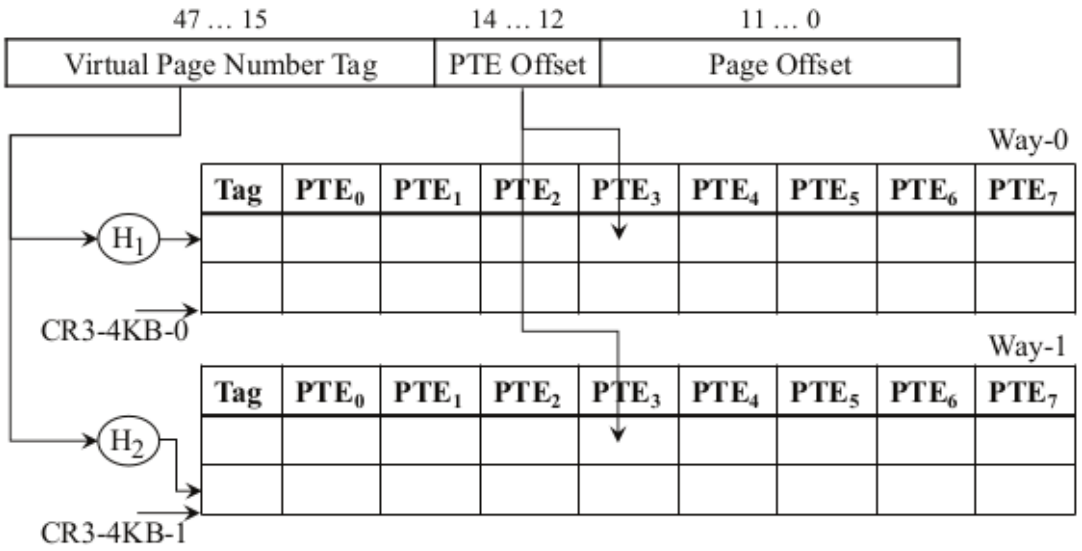

Figure 5.3: Accessing a 2 way PTE ECPT [1]

8 PTEs into a single hash table entry with a single VPN tag to improve the locality. Figure 5.3 shows how a 2 way ECPT PTE is accessed, by searching both ways using VPN tag, and then wherever tag matches, using the PTE offset bits to index into the correct page within the PTE.

5.1.1 Page table walk

New page table walk, called Cuckoo Walk, is the process of address translation using ECPT. It takes place parallelly among all ECPTs for a process, hence using memory level parallelism in address translation, unlike radix PT. To reduce the number of parallel lookups in ECPT, new page walk tables called Cuckoo Walk Tables (CWTs) are used, which contain information on which way of ECPT contains the required translation. There is a separate CWT for each ECPT, which are cached in Cuckoo Walk Caches (CWCs) to reduce their access latency. Based on information that a CWT holds, cuckoo walks are classified as shown below, and are illustrated diagrammatically in figure 5.4:

1. Complete Walk : It happens when CWT gives no information.
2. Partial Walk : CWT informs that a translation is not in a page size, so we lookup in other page sizes' ways.
3. Size Walk : CWT informs that a translation is in a particular page size.

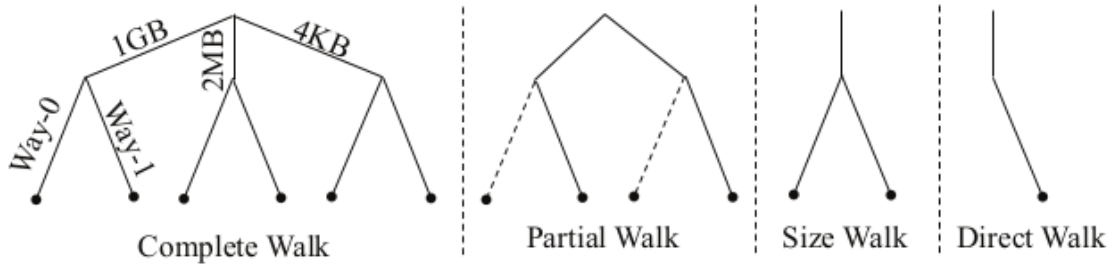

Figure 5.4: Types of cuckoo walks [1]

4. Direct Walk : CWT gives the correct page size and way of ECPT too.

5.1.2 Structure of new page walk table

CWTs store only a few bits of information, unlike their counterparts page walk caches, which store the entire VPN upto that point. Hence the size of CWTs is smaller and they have very high hit rate.

One entry of CWT contains several (64 in PMD-CWT) section headers (each section header gives information about a section, which is the virtual address space translated by one entry of ECPT). Each section header contains 4 bits of information, where first bit tells if the section maps one or more 2MB pages, second one tells if it maps one or more 4KB pages. Last two bits indicate the way of PMD ECPT that holds the mappings of 2MB pages (if present) in this section. Figure 5.5 illustrates the structure of a PMD-CWT entry.

5.1.3 Operations

5.1.3.1 Page table resize

When page table occupancy reaches a rehashing threshold (r_t, like 0.6) page table is upsized along all ways and page sizes. New page table allocated is multiplicative factor (k) times bigger than old table. Similarly, when occupancy reaches downsizing threshold (d_t, like 0.2), page table size is reduced by a downsizing factor (g).

5.1.3.2 Rehashing

To increase efficiency of a trap to OS caused on insert in PT, system rehashes one entry into new PT (or multiple if needed) on every insert operation. A rehashing

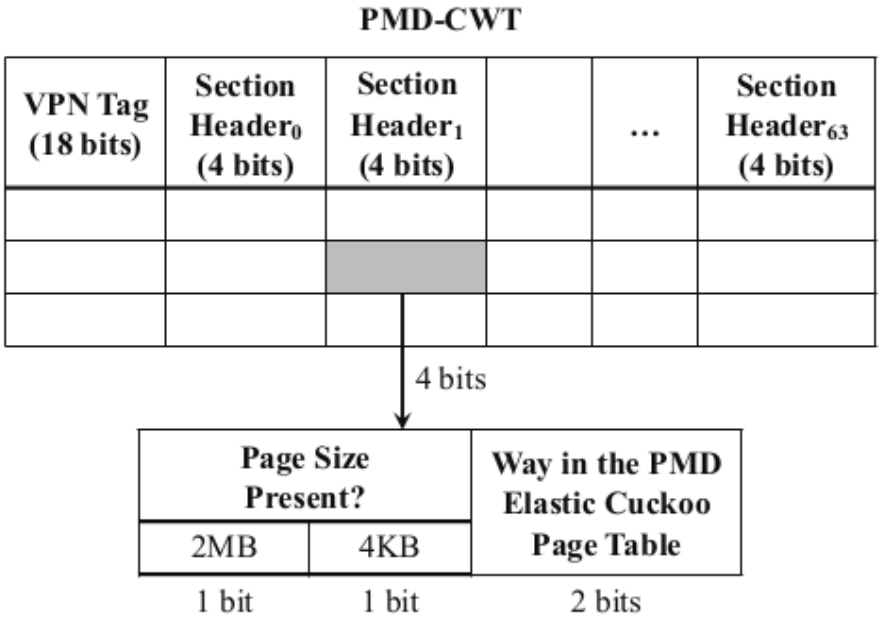

Figure 5.5: PMD-CWT entry [1]

pointer is maintained in old HPT, which points to the entry that has to be rehashed, and after rehashing, the pointer is incremented. So, old hash table is divided into two regions : Live region and Migrated region. Rehashing, as described above is shown in figure 5.6.

5.1.3.3 Look-up and delete

Look-up takes d probes in parallel for a d-ary ECPT. The needed entry is hashed and looked up in each way of ECPT using that way's hash function. If resizing is going on and the rehash pointer is greater than the index in old HPT, it means the element might be in new HPT and so it is hashed again with hash function of new HPT of same way. Else the element at that index is compared with the required entry.

Delete follows the same procedure as lookup, but it additionally deletes an element from HPT when it is found.

5.1.3.4 Insertion

Insertion of an element is attempted upto a certain number of times, after which an insertion failure is returned which triggers rehashing of entire HPT with new hash function. In each attempt to insert, a random way of ECPT is chosen and a similar

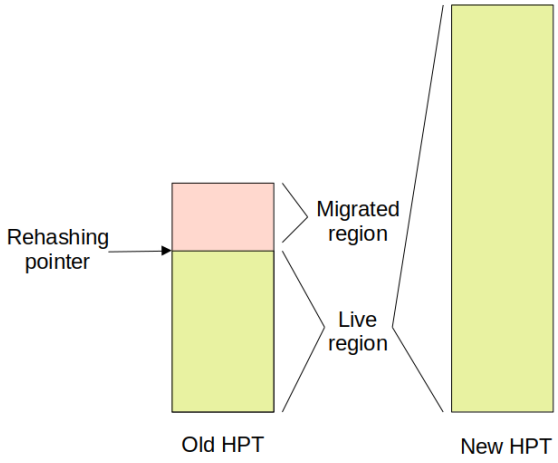

Figure 5.6: Rehashing in ECPT

procedure to look-up is followed. If the found index is empty, element is inserted there and nothing more is done. Else, current element at that place is evicted after insertion, and the evicted element again goes through these steps to get inserted in some other way of ECPT.

5.1.4 Performance analysis

ECPT provides a speed-up of 3-28% without THP and 3-18% with THP over radix page tables, as illustrated in figure 5.7.

Additionally, ECPT reduces the MMU overhead on using baseline 4KB pages by 34%, while on using THP, ECPT overhead is 41% lower than that of baseline THP.

5.2 Nested elastic hashed page table

Extending the concept of ECPT, [2] proposes nested ECPT for a virtualized setup. In a nested setup, there will be both host ECPT and guest ECPT and any address translation will pass through both of them. Host Page Table Entries (hPTEs) are tagged with host Virtual Page Number (VPN) and guest PTEs (gPTEs) are tagged with guest VPN to identify the exact entry at an index in PT upon accessing it. Accesses to all ways of ECPT in guest as well as host happen in parallel as seen in case of ECPT in a native setup.

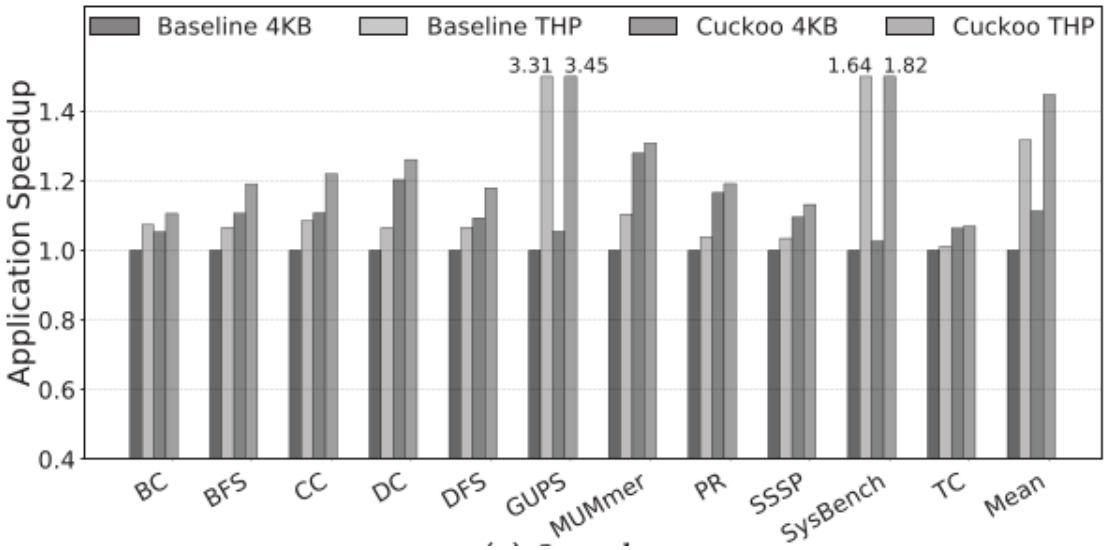

Figure 5.7: Speed-up over radix PT on using ECPT [1]

5.2.1 Page table walk

Let there be n page sizes and d ways of ECPT for each page size. A nested cuckoo walk would consist of 3 steps as described below and illustrated in figure 5.8 :

1. Guest Virtual Address (gVA) to hPTE : First the gVPN is taken and checked in all ways of all page sizes in guest for getting $n \times d$ corresponding gPTEs. Now, hPA for these gPTEs has to be obtained. For each gPTE, $n \times d$ ways in hECPT are checked, potentially making $n^2 \times d^2$ parallel accesses in hECPT. Now out of these, in each $n \times d$ hPTEs, upto one entry will have a hVPN match with the gPA that was used. But we want to match with gVPN, so we take these $n \times d$ hPTEs and execute step 2.

2. hPTE to gPA : The previous $n \times d$ entries each contain addresses of gPTEs, which are accessed and gVPN is matched. We get atmost one gPTE matching, which would contain the required gPA of the entry in gECPT of data page. Now this gPA needs to be converted into hPA.

3. gPA to hPA : Now again $n \times d$ ways of hECPT are checked in parallel and hVPN matched to get the hPTE which contains the hPA of required data page.

Overall, at worst case, $n^2 \times d^2$ parallel accesses may be required in above steps. But these ECPTs are augmented with CWTs and CWCs, which help in reducing

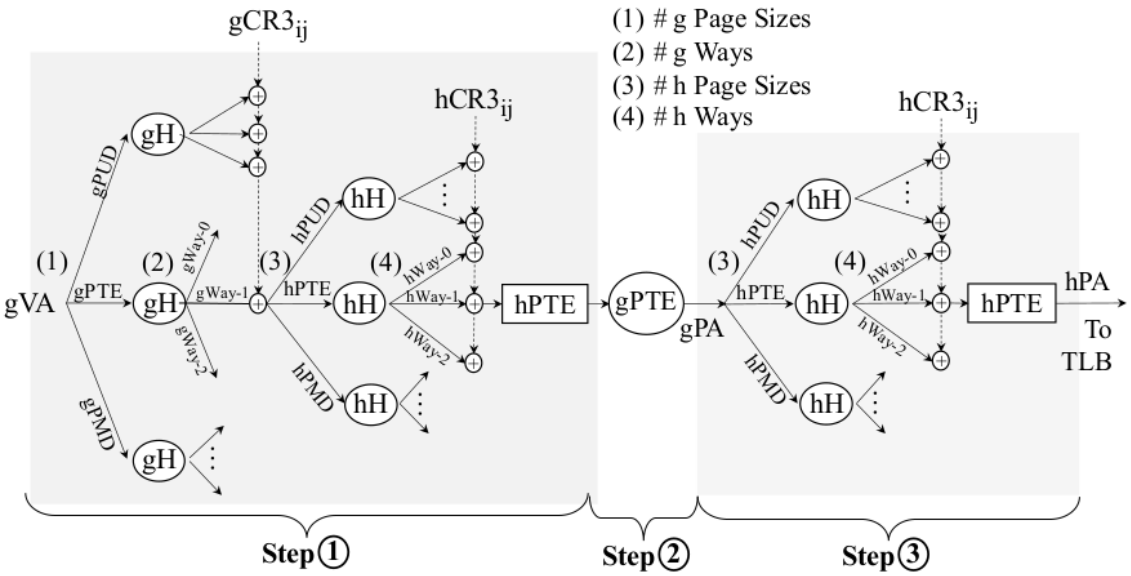

Figure 5.8: Nested Cuckoo Walk [2]

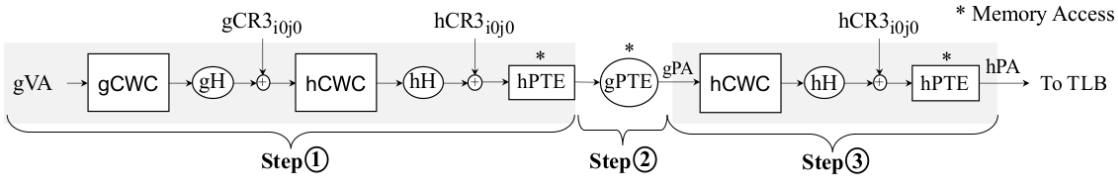

Figure 5.9: Nested Cuckoo Walk with CWCs [2]

the number of parallel accesses, and so approximately only 2.8, 2.8 and 1.6 parallel accesses are required in the above 3 steps respectively [2].

Both gCWTs and hCWTs are present and managed by guest OS and VMM respectively. gCWTs are cached in gCWCs and hCWTs are cached in hCWCs. During a cuckoo walk, first these CWCs are accessed appropriately and according to information in them, either memory accesses are not needed or fewer memory accesses may be needed. In case of CWC miss, memory accesses for all ways are needed. Nested cuckoo walk on using CWCs is illustrated in figure 5.9.

As an additional optimization, if we know that a system uses pages of only a particular size, we can optimize cuckoo walk by eliminating the factor n as only one page size is there.

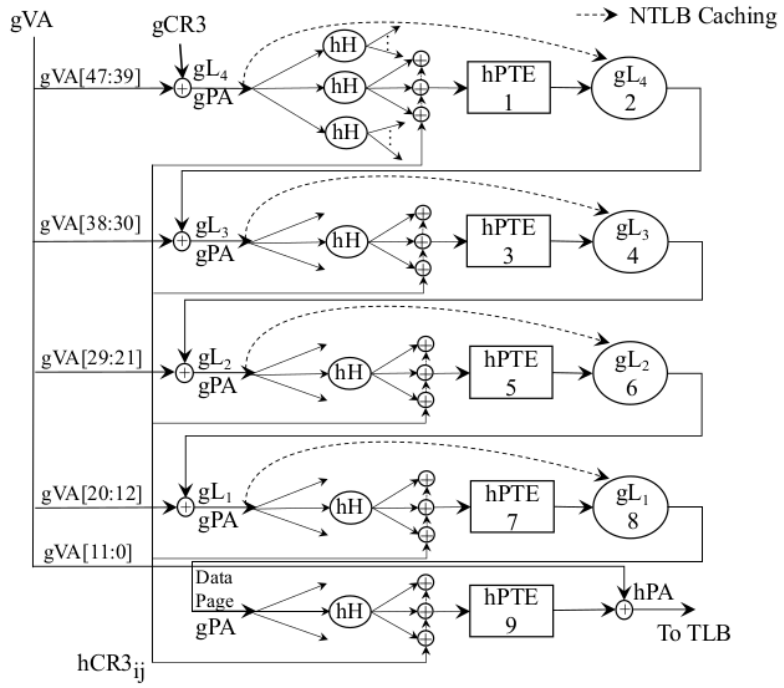

Figure 5.10: Nested Hybrid Walk with radix PT in guest and ECPT in host [2]

5.2.2 Shortcut translation cache

When there is a miss in a CWC, in addition to accessing PT, parallely the corresponding CWT entry is accessed and loaded into CWC. There is no a problem when it happens on miss in hCWC. But when this miss happens in gCWC, and when hardware tries to access the entry in gCWT, it only obtains the gPA of the entry and hence additionally needs to translate that into hPA to actually obtain that entry. To avoid this unpleasant situation of additional memory accesses, the gPA to hPA translations of entries in gCWT are cached in a small (usually 10 entry-sized is sufficient) MMU cache called Shortcut Translation Cache (STC).

5.2.3 Hybrid design

With the present hardware using radix page tables, it might be difficult to shift it to ECPT as it needs change in hardware logic and caches. Instead, guest OS could use radix PT while host uses ECPT. Such a hybrid design could support existing OS kernels, as guest OS doesn't need to change and only VMM is modified to support ECPT.

As shown in figure 5.10, a nested hybrid walk would need potentially 9 memory

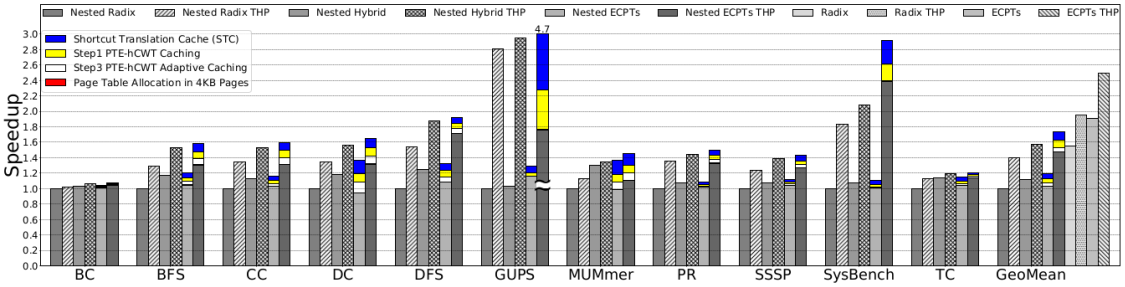

Figure 5.11: Speed-up of various applications on using nested ECPTs over nested radix PTs [2]

accesses. This is a middle-ground situation as compared to nested ECPT and nested radix PT.

5.2.4 Performance analysis

Nested ECPTs speed-up applications by an average of 1.19x as compared to nested radix PTs. Some of the major reasons for the improved performance are the ability of nested ECPTs to issue parallel memory accesses (hence making use of the available memory level parallelism) and high hit rate in STC. Performance comparison of ECPTs with radix page tables on various applications is shown in figure 5.11.

5.3 Persisting challenges

There are still some challenges remaining in the ECPT design.

The page table resizing algorithm of ECPT is not optimal. It uses an out-of-place technique where both old and new ECPTs remain in the memory while the resizing is happening. This leads to a lot of page tables being in memory in a real system scenario where multiple processes are running concurrently, and consequently ECPTs consume a lot of memory, upto 138% more memory than radix PTs as measured in [6].

Another big challenge that remains is the requirement of contiguous memory allocation to the entire page table. This is a problem that has been inherited from the hashed page table itself due to it's design. Hashed page table, by design is required to be contiguous. Also considering that ECPT can be resized, a large contiguous memory region in physical memory is needed to be allocated to ECPT whenever it is allocated or resized. It was already discussed in chapter 3 about the

Contiguous memory chunk size	CPU cycles required
4 KB	4K
8 KB	5K
1 MB	750K
8 MB	13M
64 MB	120M

Table 5.1: Analysis of time taken for allocating contiguous memory chunks of different sizes

challenges in allocating a huge page region itself. As expected, allocating a region for an entire page table is much more difficult, time taking and sometimes even fails in situations of extreme memory pressure.

Table 5.1 shows an estimate of CPU cycles required to allocate contiguous memory chunks of varying sizes, at 2GHz and 0.7 memory fragmentation (high fragmentation) in FMFI metric [6]. As we can see, it takes a large number of CPU cycles (hence large amount of time) to allocate 64MB memory chunk, which is roughly equal to the ECPT size in some applications. Also it has been found that when fragmentation is increased beyond 0.7, 64MB contiguous chunk is not possible to be allocated, therefore page table allocation fails. This is a serious problem and needs an addressal with a new idea.

5.4 Memory-Efficient hashed page table

Memory-Efficient Hashed Page Table (ME-HPT) is a new design of HPT proposed in [6], which tries to address the persisting problems mentioned in the previous section. It is built on top of existing techniques in ECPT.

5.4.1 Indirecting in page table

Instead of allocating an entire way of ECPT contiguously, each way of ECPT is broken into various chunks by introducing a level of indirection using Logical to Physical (L2P) table, as shown in figure 5.12. Let's consider chunk size to be CS. Now the address translation has two steps :

- Indexing into L2P table : First the index in L2P table which contains indirection to the required chunk of PT is found by dividing VPN by chunk size. Index in L2P table = VPN/CS

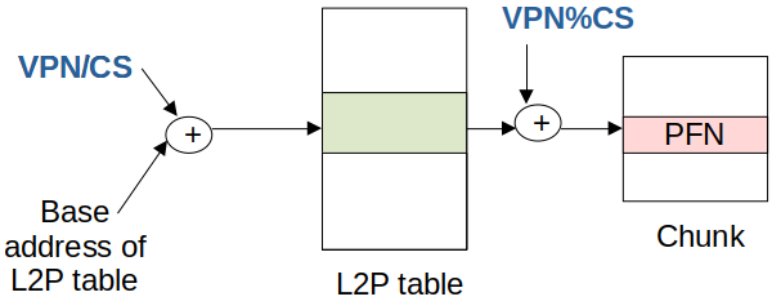

Figure 5.12: ME-HPT design with L2P table

- Accessing PTE in chunk : Now the chunk that is pointed to by L2P entry is accessed, and the required entry in chunk is obtained by modulus operation of VPN by chunk size and adding the base address of chunk to it.
  Entry in chunk = (VPN%CS) + base address of chunk

L2P table access introduces additional indirection in address translation. To avoid it, hardware can access L2P table in parallel to CWC access to generate all possible addresses to access in case of miss in CWC. This masks the L2P table latency behind CWC latency.

Context switch overhead of L2P table is quite less, as MMU points to L2P table of running process, OS only needs to switch and restore the valid entries in L2P table (only 288 entries or 1.16KB when table is full, when CS is 8KB and PA is of 46 bits) on a context switch.

5.4.2 Dynamic chunk size

ME-HPT is capable of resizing itself, but the L2P table has to be small, so it's size is kept constant. This requires CS to be dynamic by design. Four CSes of 8KB, 1MB, 8MB and 64MB have been selected. CSes have been kept in powers of 2 to simplify the division and modulo operations during address translation into shift and mask operations which are way faster.

Initially application starts with smallest CS (8KB). As the application runs and table gets filled up, more chunks of 8KB are allocated as PT gets upsized. When all 64 entries of L2P table of a way point to chunks and the PT needs to be upsized again, these 64 chunks get replaced by a single chunk of 1MB. This process then happens similarly for more upsizing, and even downsizing. HPT upsizing as described above is illustrated in figure 5.13.

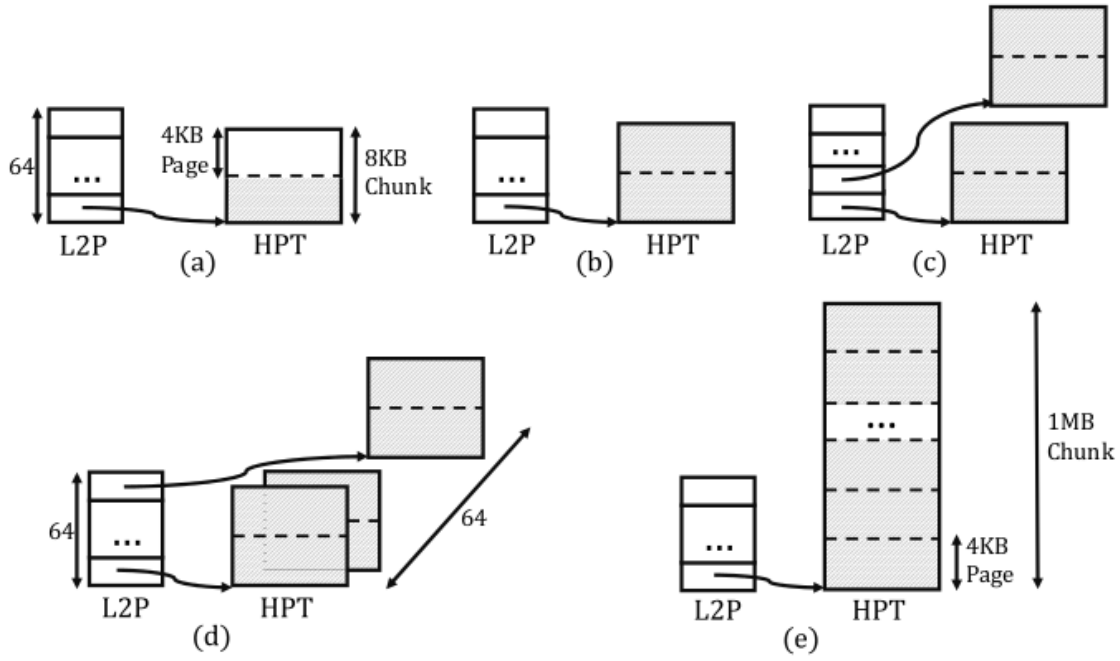

Figure 5.13: Allocation of chunks and changing chunk sizes as PT is upsized [6]

5.4.3 In-place page table resizing

In this new PT resizing approach, old and new PTs share the same memory space. Unlike in ECPT, where total memory was sum of individual memories of old and new PTs, here total memory is maximum of individual memories of old and new PTs, as shown in figure 5.14.

Same hash function is used for both old and new PT. For upsizing, the PT size is always doubled (and always halved for downsizing). So, just an extra bit of hash key compared to that used in old PT is used to index into the new PT on an upsize. Also, as the migrated region overlaps with new PT, entries may be placed into it on both insertion and rehash operations.

Unlike in ECPT where all ways were resized at same time, here resizing is done per way. OS tracks occupancy of each way. When one of the ways reaches an occupancy threshold (0.6 for upsize and 0.2 for downsize), that way is resized. To keep all ways balanced, sizes of any two ways can differ only by a factor of 2 at a time. To ensure this happens, the insertion algorithm chooses a way to insert in using a weighted random technique, which inserts elements into all ways proportional to that way's size, thus disallowing any one way to become much larger or much smaller compared to other ways.

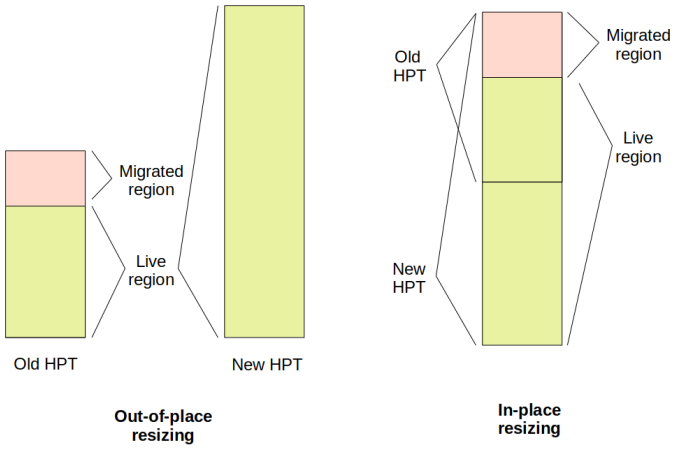

Figure 5.14: Out-of-place upsizing in ECPT vs In-place upsizing in ME-HPT

5.4.4 Operations

All operations described below are for PT upsizing scenario. They are almost similar and trivial in case of PT downsizing scenario.

5.4.4.1 Rehashing

On a rehash, an entry is moved from old PT of a way to new PT of same way. Rehashing pointer (P_i) points to the top of live region and is incremented on a rehash. As stated earlier, we use an additional bit of hash key to index into the new PT. On a rehash, if that additional bit is 0, the entry remains in the same position. Else, the entry is moved to lower half of PT at same offset as earlier in old PT. This is illustrated in figure 5.15.

In a PT downsize, two entries may collide (rehashed into same position of new PT). In such a case, one entry is cuckooed into a different way of PT.

5.4.4.2 Lookup and Deletion

During a look-up, all ways of HPT are looked up parallely. In each way, hash key obtained using old hash function is compared with P_i (if resizing of PT is happening). If the key is greater than or equal to P_i, then old hash key is used to index into old HPT as the entry might be present in old HPT. Else, new hash key is used to index into new HPT as entry might be present in new HPT.

Delete operation follows the same procedure as in look-up. In addition, it deletes the required element upon finding it.

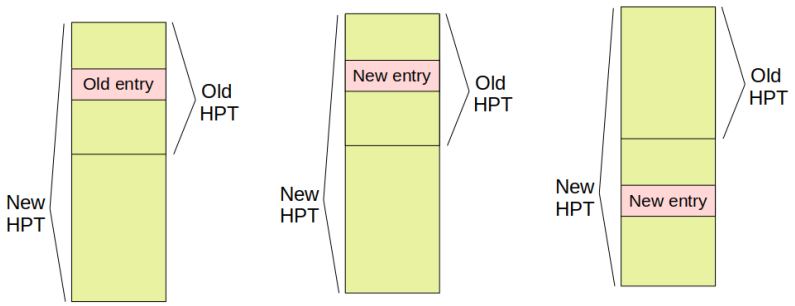

Figure 5.15: Rehashing during PT upsize with two scenarios, one where entry stays in same position and the other where entry moves to second half of new PT

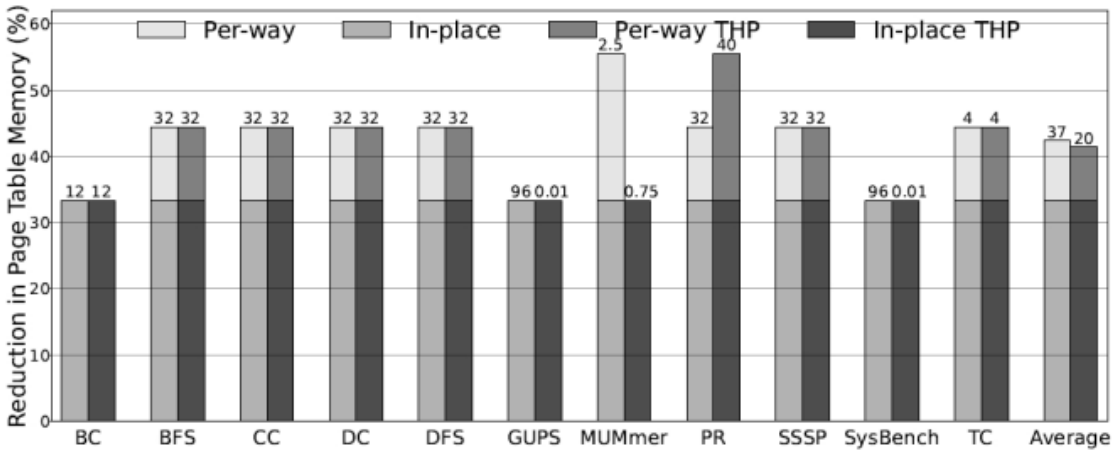

Figure 5.16: Reduction in PT size using ME-HPT as compared to ECPT [6]

5.4.4.3 Insertion

For inserting an element into HPT, one way of HPT is chosen at random. In that way, hash key obtained using old hash function is compared with P_i (if resizing is happening). If the key is greater than or equal to P_i, then old hash key is used to index into old HPT and the element is inserted at that entry. Else, new hash key is used to index into new HPT and the element is inserted at that entry instead. In case of a conflict, the existing entry is cuckooed into a different way of HPT.

5.4.5 Performance analysis

ME-HPTs provide speed-ups of upto 1.09x over ECPTs. They also reduce the maximum size of contiguous memory allocated for PT by upto 92%. As a consequence

of this, ME-HPTs use upto 43% less memory as compared to ECPTs, as shown in figure 5.16. Additionally, in-place resizing reduces number of PTEs moved during resize by upto 50%.

ECPT and nested ECPT were able to provide significant speedup over radix page table and nested radix page table respectively. In addition to that, ME-HPT was able to address the contiguous memory allocation problem in ECPT, reducing the PT size significantly while providing marginal speed-up over ECPT. Overall, address translation got a significant boost with this redesign of page table structure.

Chapter 6

Translation efficiency of NUMA systems

So far we have looked at systems with uniform memory access, i.e, all memory accesses take same amount of time. But such systems are rare in industry, where multi-threaded application servers with terabytes of main memory are used. These servers run on NUMA systems, where all memory accesses don't take same amount of time as discussed in chapter 2. NUMA systems pose new challenges to address translation that need to be addressed separately.

6.1 Placement of data pages on NUMA systems

On NUMA systems, local memory access is almost 2-4x faster as compared to remote memory access, which increases the overall memory access latency. There can be two scenarios where the effect of this is clearly visible. First is the case when a multi-threaded application is executing on multiple sockets, and there is a data page on some socket that is accessed by multiple threads. The thread that is local to that socket would be able to access the data page faster than threads on remote sockets, which increases overall application execution time. Second scenario is the one where an application was executing on a particular socket, and due to some reason, it is migrated to another socket, without migrating it's data pages. Now whenever the application needs to access a data page, remote access happens which slows down the application. So, data placement on sockets affects the performance of executing workload.

There have been previous works on optimizing data placement on such multi-socket systems, using techniques like replication and migration.

```
Level |          Socket 0          |          Socket 1          |          Socket 2          |          Socket 3
  L4  |   0 [  0   0   0   0] ( 0%) |   1 [  8   3   0   1] (75%) |   0 [  0   0   0   0] ( 0%) |   0 [  0   0   0   0] ( 0%)
  L3  |   1 [ 56  66  40  37] (72%) |   3 [ 33  43  26  26] (66%) |   0 [  0   0   0   0] ( 0%) |   0 [  0   0   0   0] ( 0%)
  L2  |  89 [11k 11k 11k 11k] (75%) | 109 [13k 13k 13k 13k] (75%) |  66 [ 8k  8k  8k  8k] (75%) |  63 [ 7k  7k  7k  8k] (75%)
  L1  | 40k [ 6M  4M  4M  4M] (67%) | 40k [ 4M  6M  4M  4M] (67%) | 40k [ 4M  4M  6M  4M] (67%) | 40k [ 4M  4M  4M  6M] (67%)
```

Figure 6.1: Page table's snapshot of Memcached [3]

Linux itself provides two data allocation policies : first touch policy, where a page is allocated local to the processor that first accesses that page, and interleaved policy, where data pages are allocated almost equally among all sockets. First touch policy may lead to most of the memory allocated in a single socket(usually when one socket's thread is initializing all memory). In addition, linux uses AutoNUMA to migrate pages among sockets to have data near the cores that are accessing it.

When a data page is being accessed from multiple sockets, data replication can be used to copy the data page to all those sockets, enabling the threads on those sockets to access the page from local memory.

But these all techniques are not used for OS data structures, like page tables, because typically OS memory is pinned in current systems. In the next section, it will be shown why page table's placement on NUMA systems is important too.

6.2 Why page table placement matters?

First, let's look at the situation of a multi-socket process, Memcached.

Figure 6.1 shows a snapshot of PT for Memcached running on 4 sockets. At each level, initial numbers (outside square brackets) show the number of pages of that level in PT belonging that socket (there is one page of level 4 in socket 1). The four columns inside square brackets show the number of pointers to next level pages to respective sockets from each socket (there are 8+3+0+1=12 pointers to level 3 pages from socket 1). The percentages in brackets show how much percentage of these pointers point to other sockets (remote sockets) from each socket. We can see that most of the pointers (page table entries) from each socket point to remote sockets. If first touch policy is used, page table is allocated mostly on a single socket, usually the socket where the memory allocator thread is running. If interleaved policy is used, then page table is evenly distributed among sockets, but at same time, if there are S sockets, then a fraction of (S-1)/S page table entries point to remote sockets. These observations show that remote page table walks happen frequently and so, address translation time increases.

Analysis on scenario of process migration has also been done in [3]. It has been

found that in comparison to case where page table is local, the case where page table is remote runs 3.3x slower when data is local, and 3.6x slower when data is also remote.

This analysis shows that page table placement has a huge impact on performance of application. This is what requires us to come up with a solution to fix this problem.

6.3 Placing page tables correctly

To solve this issue, we can migrate and replicate page tables, like data, so that remote page table accesses during page table walks can be eliminated. The same is proposed in [3] via a technique called Mitosis. Each process's PT can be dealt separately, so there may be multiple copies of PT of same process in the system. Let's look at both replication and migration of PTs using Mitosis.

6.3.1 Page table replication

To ensure that PT allocation happens smoothly, OS reserves enough memory on each socket to allocate PT pages, using page cache on each socket. While a PT is being allocated, list of target sockets is provided so that the replicas of PT are allocated on all those sockets.

Now all these PT replicas need to be kept consistent. So update on any single replica is propagated to all others. Doing this the conventional way by walking all replicas would need 4N memory accesses if there are N replicas. To optimize this, a circular linked list of replicas for each page is stored, where each page stores a pointer to next replica page, as shown in figure 6.2.

So now, for each page we need only 2 memory accesses : one to update the page and the other to read pointer to next replica page. This reduces the total number of memory accesses to 2N if there are N replicas.

PT replication has been implemented in Linux using existing functionalities. The PV-Ops interface in Linux, that's used for para-virtualization, has been used to propagate updates to pages of PT. Instead of the backend of PV-Ops, Mitosis is used as it's backend to achieve the required objective. Meta data of page is used to store pointer to next replica's page, creating a circular linked list as we saw earlier. But the access and dirty bits are set by MMU hardware, not software, hence can't be propagated. So to get value of one of those bits, all corresponding pages' bits in

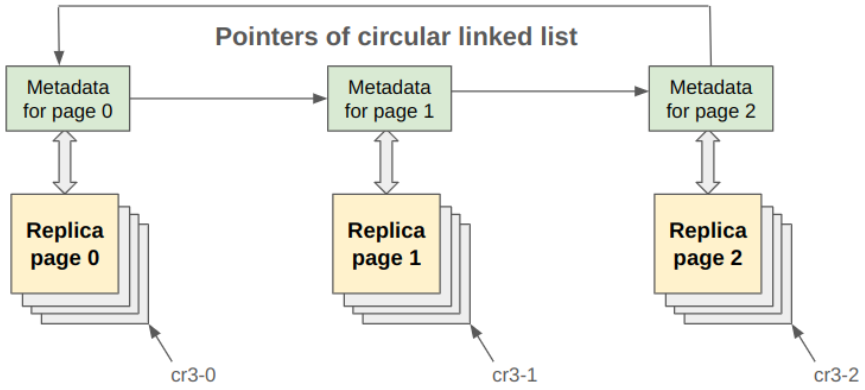

Figure 6.2: Circular Linked list to keep page table replicas consistent

all replicas are ORed so that if the bit is set in any one replica we would know.

6.3.2 Page table migration

Page table migration uses the same concept of PT replication. It replicates PT in the socket to where process is migrating. The policy regarding old replica may be customized, by either freeing it up quickly, or lazily as the memory pressure builds up in old socket.

6.3.3 System and user policies

Mitosis is mainly for long running and big memory workloads. Therefore, it's not useful for short running processes and is disabled for them. Different system wide policies like disabling mitosis for all processes, enabling per process, enabling for all processes can be set throught sysctl interface of linux.

Additionally, users can define their own set of policies according to workloads. These policies can be implemented as additional API calls to libnuma in linux.

6.3.4 Performance analysis

First let's talk about overheads. For large programs on 16 socket machine, Mitosis requires less than 2.9% of additional memory, which further reduces to 0.6% for 4 socket machine. It requires more than 20% additional memory for small programs, hence is disabled for such programs.

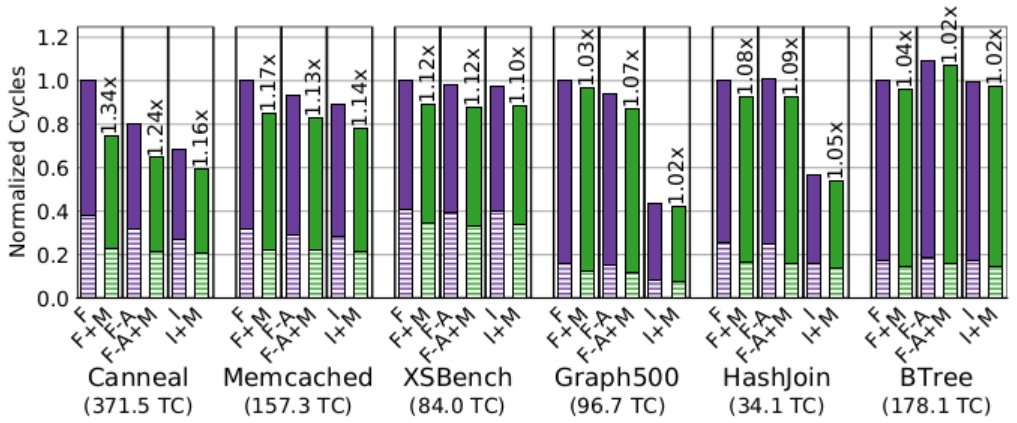

Figure 6.3: Normalized performance (with respect to 4KB first touch allocation policy) using Mitosis on 4KB pages for multi-socket processes. Mitosis run-time is shown in green color and non-mitosis in purple color. [3]

For multi-socket workloads, Mitosis provides upto 1.34x performance improvement as compared to non-replicated PT with 4KB pages. This is illustrated in 6.3.

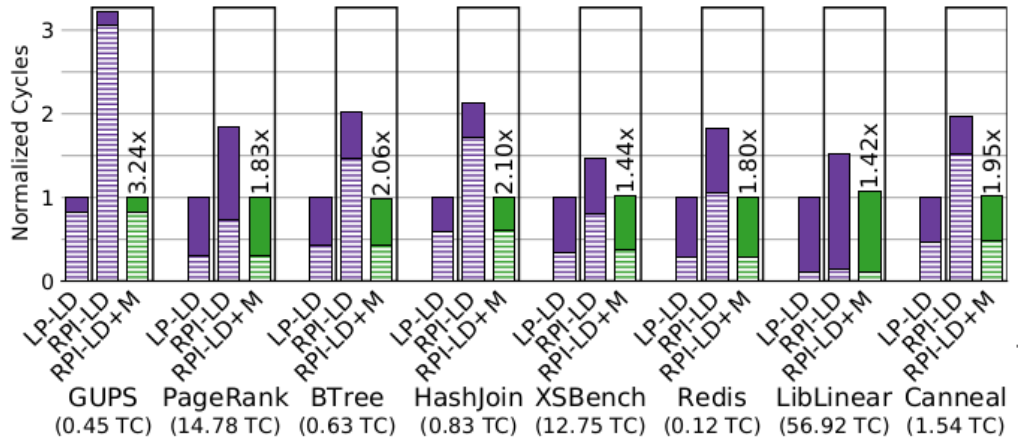

Figure 6.4: Normalized performance (with respect to 4KB first touch allocation policy) using Mitosis on 2MB pages for process migration. Mitosis run-time is shown in green color and non-mitosis in purple color. [3]

In case of workload migration, mitosis eliminates the NUMA effects of remote page table, and brings performance on par with the case of local page table, as shown in figure 6.4.

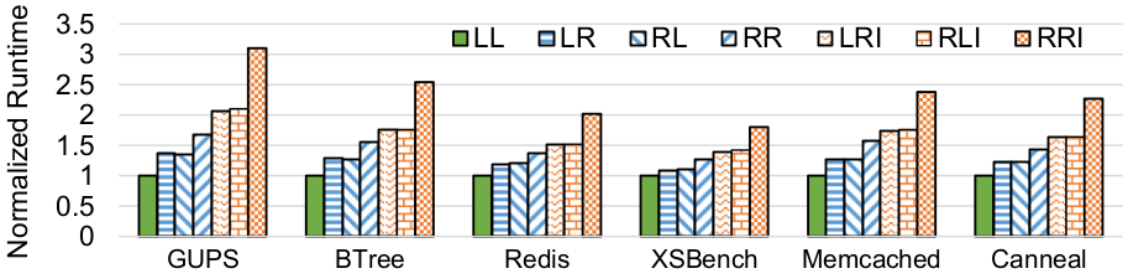

Figure 6.5: Application runtime in different scenarios of PT placement in workload migration case. First letter in a scenario denotes whether gPT is on local(L) or remote(R) socket, while second letter says the same for hPT [7]

6.4 Virtualized NUMA systems

Virtualized NUMA systems bring with them new challenges. First one is that now there are two page tables to consider: Guest PT (gPT) and Host PT (hPT). We have to apply migration and replication to both. But there is another challenge : host may or may not expose it's NUMA architecture to guest. For simplicity, let's name the VMs which have view of host's NUMA architecture as NUMA visible (NUMA-V) VMs, and those who don't as NUMA oblivious (NUMA-O) VMs. Both cases need to be handled separately.

6.5 Analyzing placement of nested page tables

Let's first look at the scenario where workload migrates from one socket to another. Here, when VMM migrates the VM, it shifts the gPT too like other memory of VM and so gPT remains local. But hPT becomes remote. Sometimes, NUMA-V VM may migrate it's workload from one virtual socket to another. In that case, gPT becomes remote and hPT may remain local or may become remote too, depending on situation. Upon measurement, it was found that either hPT or gPT being remote slows down the application by upto 1.4x. Meanwhile, when both are remote, applications may slow down by a staggering 3.1x! The exact comparison of runtime for all page table placement cases (local-local, local-remote, remote-local, remote-remote, local-remote with other workload also running, remote-local with other workload also running, remote-remote with other workload also running) is shown in figure 6.5.

Now we look at the case where a workload runs on multiple sockets and shares

same gPT and hPT. Here, we look at how many PTEs are local or remote to a socket in both gPT and hPT (PTEs are more in number by order of magnitude, so other levels of PT are ignored). A PT walk is called local if PTE is local. It was found that less than 10% PT walks were local for both gPT and hPT for NUMA-V VMs. For NUMA-O VMs, local PT walks for both PTs are almost nil. This is because guest doesn't know about the underlying NUMA architecture of host, so gPT is placed arbitrarily. Hence there is a need to focus on PT placement in virtualized NUMA systems to reduce PT walk latency.

6.6 Placing nested page tables correctly

For the workload migration scenario, it is proposed to migrate gPT and hPT effectively using vMitosis in [7].

For the multi-socket workload scenario, it is proposed to replicate gPT and hPT using vMitosis in [7]. hPT can be replicated using same concept as earlier used by Mitosis in [3]. In NUMA-V systems, as guest knows the underlying NUMA architecture, it can replicate gPT easily. But the challenge is in NUMA-O systems where guest doesn't know host's NUMA architecture. Techniques using para-virtualization (PV) and full-virtualization (FV) would be used in such a case.

6.6.1 Page table migration

In the beginning, PT pages are allocated at local NUMA socket, but additionally an array with an entry for each socket is maintained for each PT page, where each element of array tells how many PTEs of that PT page point to that particular socket. This metadata is used to decide if a PT page needs to be migrated. A PT page is said to be correctly placed if most of it's children are on same socket.

On NUMA-V system, let's say guest moves workload from one socket to another. So data will migrate, which will update PTEs of last level of PT, which will be mostly remote. This would in-turn trigger migration of last level pages of PT due to metadata that is being maintained. Similarly, when most of last level pages of PT are migrated, it's upper level starts getting migrated as PTEs at that level are mostly remote now. In this manner, entire PT gets migrated in a bottom up fashion incrementally. hPT migration also works in a similar manner in the VMM. But VMM may not be able to view guest migration in NUMA, so hPT may become misplaced. To avoid that, automatic PT migration is called occasionally.

On NUMA-O systems, it is left on the VMM to keep data and PT pages together for the guest application. gPT migration is not needed as it is automatically migrated when VMM migrates guest memory. But hPT may become remote. Here also, the above approach works and hPT gets migrated from leaf level to top level incrementally.

The above migration mechanisms are implemented on Linux using AutoNUMA, by first letting AutoNUMA fix data pages in an address range, and then updating PT counters for that address range and migrating PT pages if necessary.

6.6.2 Page table replication

The idea of Mitosis from [3] is reused to replicate hPT in both NUMA-V and NUMA-O systems. It is also used to replicate gPT in NUMA-V systems, as the NUMA architecture of host is visible to guest. But for NUMA-O systems, gPT replication needs separate mechanisms, using PV and FV.

Let's look at PV mechanism. Guest OS needs information on how many sockets the VM is allocated on, and how are the vCPUs scheduled on these sockets, to know how many gPT replicas need to be allocated and then use local gPT replica with each vCPU. Using PV, guest OS finds out socket ID for all vCPUs from VMM, to find how many gPT replicas are needed. Guest OS also relies on VMM to pin the per-socket page cache pages to the respective sockets to ensure that these page caches are locally allocated on physical server.

In FV mechanism, VMM support is not taken to migrate gPT. To find how many sockets are there on host, the fact that remote access takes more time compared to local access is used. Threads are compared pairwise for their inter-thread communication latency, and if the latency is high it means they are on different sockets, else they both are on the same socket. This helps to create virtual groups corresponding exactly to host NUMA architecture. vCPUs are assigned to these groups based on this information. To achieve gPT replication in each such virtual NUMA group, one vCPU from each group allocates one replica of PT which creates one gPT replica for each group and hence gPT replication is achieved.

6.6.3 Performance analysis

In case of workload migration, vMitosis mitigates overheads of remote PT accesses in both gPT and hPT, improving the performance to be comparable to the best case where both gPT and hPT are local to the workload. Overall, vMitosis provides upto

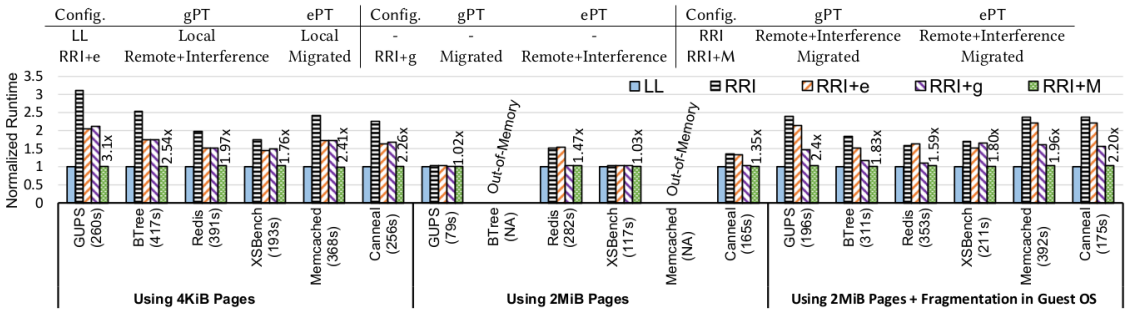

Figure 6.6: Performance of applications with and without hPT and gPT migration. Run-times are normalized to best case (both PTs local). Numbers over bars show speedup over worst case (both PTs remote) using vMitosis, for NUMA-V VM configuration. [7]

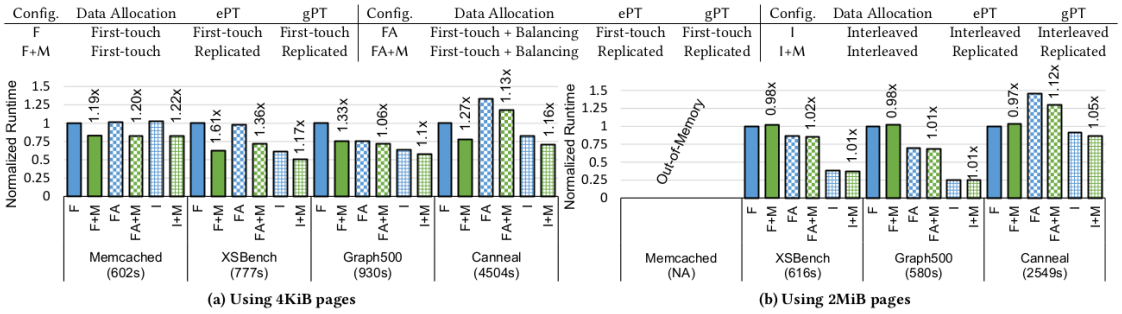

Figure 6.7: Performance of applications with gPT and hPT replication on NUMA-V systems [7]

3.1x speedup without THP and upto 2.4x speedup with THP. This is illustrated in 6.6.

In case of multi-socket workload, PT replication with vMitosis provides upto 1.6x speedup on NUMA-V systems as shown in figure 6.7, and upto 1.4x speedup on NUMA-O systems, as illustrated in figure 6.8.

It can be observed that performance gain is not significant with 2MiB huge pages, since using huge pages itself reduces TLB misses. But using huge pages may sometimes lead to application failure due to unavailable memory, as we can see in case of Memcached.

Overall, Mitosis and vMitosis were able to eliminate the NUMA effects of page table walk wherever it was possible. This whole discussion shows that page table placement on NUMA systems affects the memory address translation significantly, just like data placement on NUMA systems does.

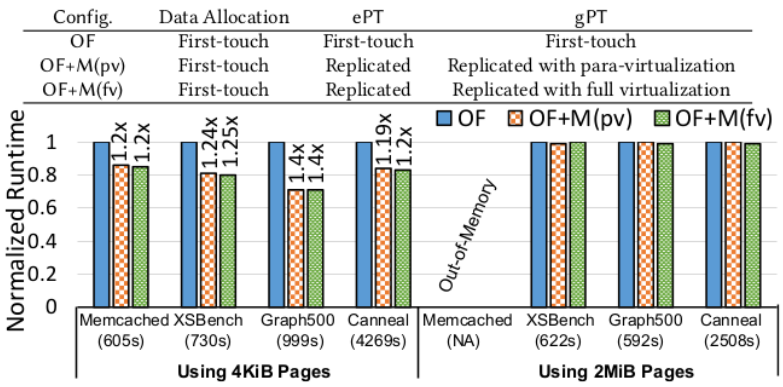

Figure 6.8: Performance of applications with gPT and hPT replication on NUMA-O systems [7]

Chapter 7

Conclusion and Future Work

We started the discussion by understanding what is memory address translation and why is it needed. Then we had a look at current address translation mechanisms and tried to reason why their efficiency needs to be improved. Then we began exploring various ways of improving the address translation.

We saw the approach to reduce TLB misses to avoid page table walk itself. Though we saw performance improvements with this approach, soon realization dawned upon us that TLB is not scalable and reducing TLB misses is not enough given the ever increasing memory footprint of applications. We then explored ways of making the page table walk itself efficient. We saw the approach to reduce page table levels, especially in a virtualized environment. Performance improved marginally, and then not satisfied with this, we realized that the problem lies with the level-wise organization of radix page table which makes utilization of memory level parallelism in page table walk impossible. Then we changed our course by exploring redesign of page table on the lines of hashed page table. We realized hashed page table's limitations due to collisions and non-efficient resizing, and looked at elastic cuckoo page table which tries to fix them. We also looked at extension of this approach to virtualized environment by using nested elastic cuckoo page table. We found both elastic cuckoo page table and it's nested version to provide significant improvements over radix page table and it's nested versions respectively. Then we discussed about the challenges persisting in their designs which demands for the entire page table to be allocated contiguous physical memory, which is difficult and sometimes impossible. To fix this, we looked at memory efficient hashed page table which breaks the page table into chunks that fixes the contiguous memory requirement problem and consumes significantly less memory as compared to elastic cuckoo page table.

Then we moved towards non uniform memory access (NUMA) systems, which are prevalent in places like data centers and talked about how page table placement significantly affects memory address translation efficiency. We argued that page table replication and migration improves application performance and saw how Mitosis accomplishes it. We also had a look at virtualization in NUMA systems and how it is challenging to replicate and migrate page tables in such a setup due to additional complexities associated with the fact that hypervisor may or may not expose it's NUMA topology to guest. We also saw how vMitosis tries to tackle that problem.

For future work, we can think about combining multiple approaches to see if performance improves further. For example, it is worth exploring how would elastic cuckoo page table and memory efficient hashed page table perform on NUMA system, and how can we replicate and migrate them efficiently as the approaches used to do so for radix page table might not work here. We can further extend such an exploration to virtualized NUMA system too.

Bibliography

[1] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1093–1108, 2020.

[2] J. Stojkovic, D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Parallel virtualized memory translation with nested elastic cuckoo page tables," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 84–97, 2022.

[3] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, "Mitosis: Transparently self-replicating page-tables for large-memory machines," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 283–300, 2020.

[4] A. Manocha, Z. Yan, E. Tureci, J. L. Aragón, D. Nellans, and M. Martonosi, "Architectural support for optimizing huge page selection within the os," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1213–1226, 2023.

[5] W. Jia, J. Zhang, J. Shan, Y. Du, X. Ding, and T. Xu, "Hugegpt: Storing guest page tables on host huge pages to accelerate address translation," in *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 62–73, IEEE, 2023.

[6] J. Stojkovic, N. Mantri, D. Skarlatos, T. Xu, and J. Torrellas, "Memory-efficient hashed page tables," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1221–1235, IEEE, 2023.

[7] A. Panwar, R. Achermann, A. Basu, A. Bhattacharjee, K. Gopinath, and J. Gandhi, "Fast local page-tables for virtualized numa servers with vmito-

sis," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 194–210, 2021.